Universidad Zaragoza
1542

Trabajo Fin de Grado

Capa de Acceso al Hardware para microcontroladores ARM Cortex M

Hardware Access Layer for ARM Cortex M microcontrollers

Autor

Pablo Escribano García

Director

Isidro Urriza Parroqué

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2020

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe entregarse en la Secretaría de la EINA, dentro del plazo de depósito del TFG/TFM para su evaluación).

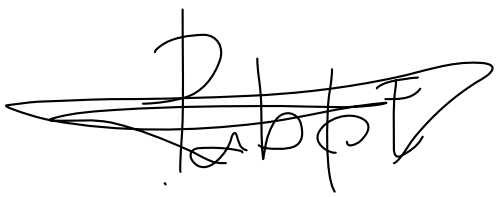
D./Dª. Pablo Escribano García ,en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza, Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado

(Título del Trabajo)

Capa de Acceso al Hardware para microcontroladores ARM Cortex M

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 26 de Junio de 2020



Fdo: Pablo Escribano García

Capa de Acceso al Hardware para microcontroladores ARM Cortex M

Resumen

El presente proyecto consistirá en el diseño una Capa de Acceso al Hardware (HAL en inglés que proporcione a desarrolladores de sistemas embebidos una serie de funciones de configuración y control de los principales periféricos que se encuentran en los microcontroladores actualmente.

El diseño de la HAL deberá ser portable entre plataformas, por lo que será desarrollada en un lenguaje de programación y con unas librerías que ya estén consolidadas en la industria y que sean reconocidas por la mayor parte de los entornos de desarrollo actuales.

Se desarrollará una Interfaz Gráfica de Usuario para facilitar el aprendizaje de manejo de la HAL. Esta interfaz permitirá a los programadores utilizar las funciones de configuración de cada periférico que se encuentran implementadas en la HAL de una forma gráfica, sin tener que escribir ninguna línea de código.

Además, dado el creciente uso de los procesadores ARM Cortex en todos los ámbitos de la Industria se va a trabajar con dos microcontroladores con procesador ARM Cortex M para verificar que la HAL es capaz de llevar a cabo el trabajo para el que ha sido creada y que no presenta errores de diseño.

Por último, las herramientas desarrolladas a lo largo del proyecto pretenden ser intuitivas y de fácil manejo, por lo que se elaborará una documentación clara y concisa sobre las implementaciones realizadas. Para ello, se hará uso de un generador de documentación muy extendido en el ámbito industrial y de fácil implementación llamado Doxygen con el que las funcionalidades de la HAL quedarán correctamente ilustradas, mejorando el entendimiento del código.

Hardware Access Layer for ARM Cortex M microcontrollers

Abstract

The current project will consist in the design of a Hardware Access Layer that will bring for embedded system developers, configuration and control functions for the main peripherals that is possible to find in a microcontroller.

The HAL design has to be portable between platforms, that is why it will be developed in a programming language and with libraries that have been tested in the industrial sector during all these years and are currently recognised by the majority of IDE.

A Graphical User Interface will be also developed in order to ease the learning process of the HAL. This interface will allow programmers to use the configuration functions that have been implemented in the HAL in a graphical way, without writing any piece of code.

In addition, due to the increasing use of ARM Cortex processors in the Industry, the HAL is going to be tested with two microcontrollers that have inside them a processor like the one commented before.

Finally, for the tools developed during the whole Project that aimed to be intuitive and user-friendly, a crystal clear documentation will be produced. In order to do that, Doxygen, an extended documentation generator Will be used, showing how the functions and other implementations have been made.

Índice

Tabla de Figuras

Capítulo 1: Introducción

1.1 Organización del proyecto

Esta memoria se encuentra organizada en una serie de ocho capítulos en los que se detallan las labores realizadas para alcanzar los objetivos planteados en este Trabajo de Fin de Grado.

En el capítulo 1, se tratan los objetivos que se van a abordar, la metodología seguida y la planificación a seguir, se hace un pequeño inciso en el contexto histórico en el que se encuentra la tecnología con la que se ha trabajado y el alcance al que se aspira llegar.

En el capítulo 2, se profundizará en el trabajo realizado, tratando los aspectos fundamentales para entender que es una HAL y cómo funciona, las herramientas utilizadas en el desarrollo de esta y el proceso de verificación y chequeo de errores que se ha llevado a cabo para justificar la viabilidad técnica de la misma.

En el capítulo 3, se abordarán todos los temas referentes al desarrollo de la Interfaz Gráfica, incidiendo en sus características y la estructura de esta.

En el capítulo 4, se mostrará el subproyecto realizado como demo para demostrar un ejemplo del correcto funcionamiento de la Interfaz Gráfica y de la HAL, analizando los componentes utilizados y la aplicación móvil elaborada.

En el capítulo 5, se detallan las conclusiones que se han extraído a lo largo del trabajo contrastándolas con los objetivos planteados en el primer capítulo.

En el capítulo 6, se analizan posibles vías de investigación para conseguir una mayor amplitud del proyecto.

En el capítulo 7, se encuentran las referencias utilizadas para la elaboración de la memoria.

En el capítulo 8 aparece información complementaria en forma de Anexos a la que se alude desde la memoria en caso de ser preciso conocer más información referente a alguno de los puntos tratados.

1.2 Objetivos

1. Desarrollar en lenguaje C de una capa de acceso al Hardware (HAL) portable para procesadores ARM Cortex-M.

2. Crear una Interfaz gráfica que facilitar el acceso a la HAL y que se adapte gráficamente a las características del microcontrolador de trabajo.

3. Elaborar una documentación clara de la HAL.

4. Aplicar y trabajar con los estándares utilizados hoy en día en las herramientas utilizadas (Keil, Doxygen, lenguaje C, Python) para el correcto desarrollo de la HAL.

5. Hacer uso de los buenos hábitos de programación en las implementaciones de la HAL y de la Interfaz Gráfica.

6. Verificar que no existen errores de diseño que den lugar a fallos en el futuro que afecten a aquellos programadores que hagan uso de las herramientas planteadas.

7. Elaborar una demostración que ilustre el correcto funcionamiento de la HAL; es decir, muestre un ejemplo en el que se utilicen las mismas funciones, declaradas la HAL, para dos microcontroladores de diferentes fabricantes y se consiga el mismo comportamiento.

1.3 Planificación y Metodología

La metodología planteada tiene por objeto la programación de cada uno de los pasos que se van a dar en el proyecto de tal forma que estén bien justificados y en línea con la consecución del objetivo final.

Todas estas tareas serán planificadas y distribuidas a lo largo del tiempo (desde el inicio del TFG el día 10 de febrero hasta la fecha de entrega el viernes 26 de junio del año 2020) resultando en una holgura libre de 13 días para posibles imprevistos para el trabajo de desarrollo del software, como se puede observar en la Figura 1.

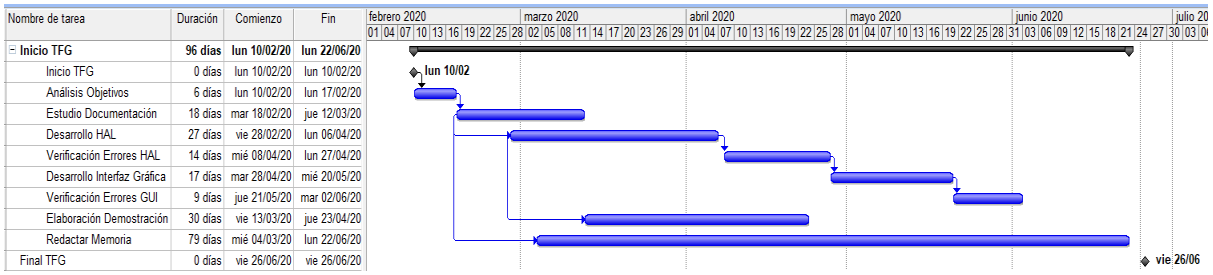


Figura 1. Diagrama de Gantt de las tareas a realizar.[24]

La metodología propuesta contendrá las siguientes tareas:

1. **Análisis de los objetivos:**

   Inicialmente, se realizarán un análisis de los objetivos que se desean abordar para planificar el inicio del estudio de cada uno de ellos.

2. **Estudio de la documentación existente:**

   Seguidamente, se llevará a cabo un estudio previo de los microcontroladores MSP432P401R del fabricante Texas Instruments [6] y del microcontrolador RSL10 del fabricante ON Semiconductor [8] a través de las hojas de características proporcionadas por los fabricantes. Estos microcontroladores cuentan con un procesador ARM Cortex M [22] y serán utilizados para justificar la viabilidad técnica de la HAL.

   A continuación, se investigará acerca de cómo son implementadas actualmente las Capas de Acceso al Hardware mediante la lectura del libro "Reusable Firmware Development" de Jacob Beningo [1]. Se analizarán cuáles son las reglas y los buenos hábitos de la programación en C según los contenidos del libro "Zero Bugs… Embedded C Coding Standard" de Michael Barr [2]. Se abordará qué son y cómo funcionan los procesadores Cortex ARM M a través de "The definitive guide to ARM Cortex M3 and Cortex M4 processors" de Joseph Yiu. [3]

   El libro de Jacob Beningo, "Reusable Firmware Development" también contiene un capítulo dedicado a realizar una buena documentación de código haciendo uso de Doxygen que será utilizado este libro como soporte para el manejo del generador de documentación.

3. **Desarrollo de la HAL**

   Una vez adquiridos todos estos conocimientos se procederá a desarrollar la HAL en el entorno de desarrollo Keil. Los primeros pasos con la plataforma consistirán en utilizar código desarrollado por otros programadores para familiarizarse con el entorno y pasar seguidamente a trabajar con la HAL.

4. **Verificación de errores**

Una vez finalizada la HAL, esta se pondrá a prueba con los dos microcontroladores con procesador ARM Cortex M con el objetivo de detectar y subsanar aquellos errores cometidos en el proceso, así como documentar todo el código.

5. **Desarrollo Interfaz gráfica**

La siguiente etapa del proyecto consistirá en la formulación de la Interfaz Gráfica en el editor de documentos Sublime. En Sublime se trabajará con [11] y sus librerías de las cuales ya se parte con un conocimiento previo. Por supuesto este software también será sometido a un testeo y a un proceso de corrección de errores.

6. **Demostración**

Finalmente, para justificar la viabilidad técnica de la herramienta se elaborará un programa que permita a ambos microcontroladores comunicarse vía Bluetooth con una aplicación de móvil de elaboración propia desarrollada en AppInventor. Esta aplicación enviará una serie de comandos al microcontrolador de forma inalámbrica que le indicarán al microcontrolador si encender o apagar una lámpara o encender uno de los tres leds (rojo, verde y azul) con los que cuenta el microcontrolador.

Esta prueba se realizará con los dos microcontroladores antes mencionados, cargando en ellos un programa que use las mismas funciones (implementadas en la HAL) para llevar a cabo tales acciones. De esta forma se podrá verificar que las funciones utilizadas para el código subido en ambos microcontroladores, son las mismas sin importar cual es el microcontrolador que se esté trabajando.

1.4. Historia y Estado del Arte

1.4.1 HAL

Desde las pasadas décadas los sistemas embebidos se han visto incrementados en complejidad y en el número de características que presentan, pero no ha sucedido lo mismo con el tiempo que se precisa para desarrollar códigos que las aprovechen. Esto ha obligado a muchos desarrolladores a reutilizar antiguos programas y a invertir tiempo para articular la forma para acelerar el proceso. [1]

Con el paso de los años y debido al aumento en la capacidad de memoria disponible en los microcontroladores (lo cual era un gran impedimento para haber planteado una solución como la que se presenta en este proyecto) surgen las Capas de Acceso al Hardware.

Estas HAL definirían una estructura organizada de capas que una vez programadas, tarea que requería de mucho tiempo y esfuerzo, pero que una vez desarrollada permitían configurar cualquier microcontrolador mucho más rápido, disminuyendo los tiempos que tenían que ser dedicados.

En la actualidad, estas capas no son tan utilizadas como se podría suponer, ya que al requerir de un elevado tiempo para ser elaboradas y en muchas ocasiones se prefiere hacer código exclusivo para el microcontrolador que se está utilizando en ese momento.

No obstante, actualmente existen empresas especializadas en este ámbito no dudan en invertir este tiempo inicial para conseguir trabajar más cómodamente en el futuro. Esta situación tiene el problema de que al final las HAL son propias de cada empresa, donde cada una le aporta su estilo y no da lugar a elaborar unos estándares universales que determinen cómo tiene que llevarse a cabo el proceso.

Hoy en día, el conocimiento que se puede adquirir sobre el proceso de diseño de una HAL es amplio, pero con el inconveniente de no existir un criterio unificado sobre cómo realizarlas exigiendo al desarrollador que está aprendiendo, a ser capaz de extraer lo mejor de cada una.

Además, a lo largo de estos últimos años los fabricantes de las nuevas versiones y familias de microcontroladores no modifican los periféricos con cada versión por lo que una vez desarrollados el software para cada periférico de un fabricante estos podrán volver a ser usados muy probablemente en las futuras versiones del microcontrolador, con el consecuente ahorro en tiempo.

1.4.2 ARM Cortex

La arquitectura ARM (Advanced RISC Machine) es un conjunto de instrucciones de 32 y 64 bits para ordenadores desarrollada por la empresa ARM Holdings.
Esta arquitectura respecto a las anteriores existentes en el mercado lograba, en 1893, una reducción de costes, calor y energía lo cual con sus sucesivas mejoras y actualizaciones daría lugar a que, en 2005, alrededor del 98% de los más de mil millones de teléfonos móviles vendidos utilizaban al menos un procesador ARM. [23]

Actualmente, esta arquitectura se puede encontrar desde en teléfonos inteligentes, hasta microcontroladores pasando por tabletas, relojes inteligentes, videoconsolas portátiles, calculadoras, reproductores digitales multimedia etc.

El negocio principal de ARM Holdings es la venta de núcleos IP (propiedad intelectual), que son utilizadas para crear microcontroladores y CPU basados en este núcleo [23]. De hecho en la actualidad existen tres series de procesadores (A, R y M) que incorporan esta arquitectura y se emplean con finalidades diferentes.

Así, los procesadores ARM Cortex– A son los utilizados para aquellas soluciones electrónicas que se vean obligadas a realizar tareas complejas como controlar múltiples aplicaciones o poder soportar un sistema operativo. Son los usados principalmente en los teléfonos inteligentes, tabletas etc. [20]

Los ARM Cortex – R son los utilizados para llevar a cabo tareas de procesado en tiempo real y que requieren de cierta velocidad en las operaciones. [21]

Finalmente, los ARM Cortex M son los procesadores que se implementan en los microcontroladores que se encuentran en el ámbito industrial, ya sea para realizar aplicaciones del Internet de las cosas, o proyectos que requieran bajos consumos, altavoces etc. [22]

Esta expansión que han logrado en los últimos años hace que su estudio tenga un especial interés y es por ello que han sido elegido microcontroladores que incorporan un procesador con arquitectura ARM Cortex M como unidades de trabajo para el desarrollo de la HAL.

1.5. Alcance

Este proyecto pretende construir una HAL con un estilo orientado a la facilidad de uso, haciendo un menor hincapié en la optimización de recursos, la funcionalidad, la portabilidad entre entornos de desarrollo y la claridad de la documentación aportada.

Se implementará una Interfaz Gráfica de Usuario que sea adaptable según las características del microcontrolador con el que esté trabajando el programador. Esta Interfaz Gráfica deberá permitir una

fácil gestión de la HAL generando tras su interacción con el usuario, los archivos necesarios para la configuración de los periféricos del microcontrolador que el usuario desee.

Tanto para la Interfaz Gráfica como para la HAL van a ser diseñadas para trabajar con los periféricos que más comúnmente se encuentran en los microcontroladores actuales; estos son, los pines digitales (DIO), los módulos de comunicación serie (UART, SPI, I2C), los Temporizadores y los conversores analógico-digitales (ADC) dejando una vía abierta a incorporar más periféricos en un futuro.
También, se procurará ofrecer una documentación detallada sobre las definiciones y el software generado con objeto de disminuir la curva de aprendizaje de los mismos.

Por último, se pretende hacer uso de únicamente protocolos y lenguajes de programación estandarizados y consolidados en el ámbito ingenieril, logrando una mayor accesibilidad de la herramienta al público.

Capítulo 2: HAL

Una HAL (Capa de Acceso al Hardware de sus siglas en inglés “Hardware Abstraction Layer”) se define como una serie de capas de programación que permiten a los programadores controlar y configurar el Hardware de un dispositivo electrónico. Estas capas de programación que constituyen la HAL interactúan entre ellas, compartiendo información y recursos para conseguir nuevas funcionalidades.

2.1 Programación por capas

La programación por capas constituye un modelo en el desarrollo de software en el que se busca que todos los códigos que componen un sistema software se encuentren desacoplados entre ellos, evitando la dependencia de unos en otros. De esta forma, en caso de requerirse algún cambio en el funcionamiento del software o de aparecer algún error en el mismo, este solo afectará a la capa correspondiente evitando tener que revisar el código fuente de otros módulos para corregir el error. [4]

La HAL presentada está constituida por dos capas, la capa de Middleware y la capa de Driver como se puede ver en la Figura 2. Estas dos capas interactúan con otras dos capas que serán aportadas por los desarrolladores que hagan uso de la HAL, llamadas capa de Aplicación y capa de Hardware.

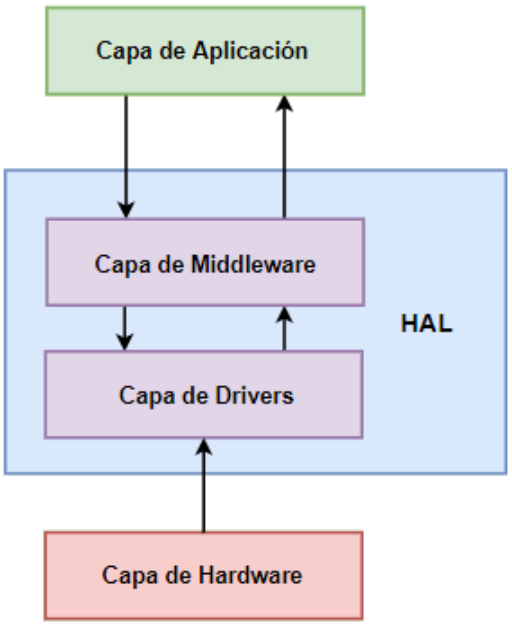


Figura 2. Capas de las que consta la HAL (Middleware). *[24]*

El objetivo principal de la HAL es conseguir que cuando se decida cambiar el microcontrolador de trabajo por uno nuevo, todas aquellas funciones que servían para controlar el antiguo microcontrolador, funcionen también con el nuevo, siendo solo necesario modificar algunas de las capas de la HAL y otras no tengan que ser modificadas.

Esto se consigue haciendo que las capas que no tienen que cambiar nunca, no contengan información referente a un microcontrolador en concreto, dejando esta información en capas inferiores con las que se comunicarán. De esta forma, se consigue que, aunque cambie el microcontrolador, no lo hagan los

nombres de las funciones utilizadas con el anterior microcontrolador y por lo tanto se acelera el desarrollo de aplicaciones. En la HAL aquí presentada, el usuario solo deberá modificar la capa de Hardware y completar algunos espacios de código de la capa de Drivers para empezar a controlar un nuevo microcontrolador.

2.2 Interfaces de la HAL

2.2.1 Interfaz en programación

Dentro de cada una de las capas antes mencionadas, se definen un grupo de subrutinas, funciones y procedimientos, con fines comunes que facilitan el trabajo con microcontroladores. Cada uno de estos conjuntos de instrucciones es lo que se conoce como Interfaz. Ver en la referencia [1](Capítulo 1: "Concepts for Developing Portable Firmware").

Las interfaces por lo tanto contienen herramientas que permiten a los desarrolladores construir sus aplicaciones. Las interfaces no están solo sujetas a la programación de microcontroladores, sino que pueden ayudar a realizar cualquier actividad que precise de software como; por ejemplo, las labores de estilismo en una página Web.

En el caso de una HAL, las interfaces están especializadas en el acceso a los recursos Hardware de los microcontroladores. Estos recursos son habitualmente conocidos como periféricos.

Además, las interfaces suelen incorporar un conjunto de códigos de estado para informar al programador del resultado de las operaciones que han llevado a cabo, permitiendo al programador tener conocimiento sobre si ha existido algún problema o por el contrario todo ha ido correctamente.

2.3 Periféricos

Los periféricos son todos aquellos circuitos digitales que permiten al microcontrolador interactuar con el mundo "exterior" y van desde módulos para habilitar/deshabilitar salidas digitales hasta circuitos que permiten al microcontrolador comunicarse con el exterior. [5]

Las interfaces son todas las herramientas utilizadas para trabajar con estos periféricos permitiendo a los desarrolladores configurarlos y utilizarlos como deseen.

Los periféricos son diferentes según cada fabricante de microcontroladores por lo que se requiere de tener un conocimiento específico sobre cada uno de ellos (tanto por parte del desarrollador que haga uso de la HAL como por parte del programador de la HAL) para trabajar con ellos adecuadamente.

El desarrollador de la HAL debe tener un conocimiento amplio sobre las propiedades generales de los periféricos de cada fabricante para ser capaz de encontrar las características comunes que tienen entre ellos y así elaborar unas interfaces lo más flexibles posible que permitan integrar en la HAL al mayor número de microcontroladores posible.

Por otro lado, el programador que hace uso de la HAL debe conocer las características más específicas de su microcontrolador para exprimir al máximo las posibilidades que este le ofrece.

Es por ello que para la construcción de la HAL h que lleva sido necesario llevar a cabo un estudio de diferentes microcontroladores para adquirir una visión global de las propiedades más comunes del periférico del mercado.

Parte de esta visión general ya había sido adquirido previo al desarrollo del documento, pero se ha decidido completar mediante el análisis de dos microcontroladores y que integran un procesador ARM Cortex M como son el RSL10 y MSP432P401R.

2.3.1 MSP43201R

2.3.1.1 Introducción

El MSP43201R [6] es un microcontrolador producido por Texas Instruments que pretende ser utilizado tanto en el ámbito de la enseñanza como en el industrial. Este microcontrolador destaca por su procesador, un ARM Cortex M4 de 32 bits que permite trabajar en tiempo real y que permite una sencilla gestión de las interrupciones gracias a un Nested Vectored Interrupt Controller (NVIC).

2.3.1.2 Periféricos del MSP43201R

Los principales periféricos de este microcontrolador son los siguientes:

1. **Procesador**: El procesador principal es un ARM Cortex M4F que trabaja a 48 MHz y cuenta con una unidad de FPU que le permite trabajar con números enteros de 32 bits.

2. **Memoria**: Cuenta con una memoria flash de 256 KB, 64 KB de SRAM y 32 KB de ROM.

3. **Relojes y osciladores:** Cuenta con sistema de relojes configurable por software que permite al desarrollador modificar la frecuencia de trabajo de cada reloj. Dentro de este sistema destaca principalmente el Oscilador Controlado Digitalmente (DCO) que puede llegar a trabajar con frecuencias de hasta 48 MHz.

4. **ADC:** Dos conversores Analógico - Digital de bajo consumo. Puede trabajar con una frecuencia de muestreo de hasta 10 ^6 muestras/segundo y cuenta con 24 canales de entrada.

5. **DIO:** Los DIO son las salidas digitales que interactúan con "el exterior" a través de pines que configurados por software. Trabajan entre 0 voltios a 5 voltios respecto de la referencia interna del microcontrolador.
   Todos estos pines digitales cuentan con un sistema que garantiza que las pérdidas de energía por corrientes de fugas sean mínimas (20 nA como máximo). En el MSP43201R hay 11 puertos con 8 pines digitales en cada uno.

6. **Interfaces de comunicaciones serie, SPI, I2C y UART:** Cuenta con dos periféricos llamados eUSCI_A y eUSCI_B que integra 4 módulos en cada uno que permiten ser configurados para funcionar como interfaces SPI, I2C y UART.

7. **Temporizadores:** El MSP43201R incorpora 4 temporizadores de 16-bit y 2 temporizadores de 32-bit para contar los ciclos del reloj que los alimente y permitir al programador tener control del tiempo. Todos son controlados digitalmente.

Se pueden apreciar todos estos periféricos en el diagrama de bloques que proporciona el fabricante en la Figura 3.

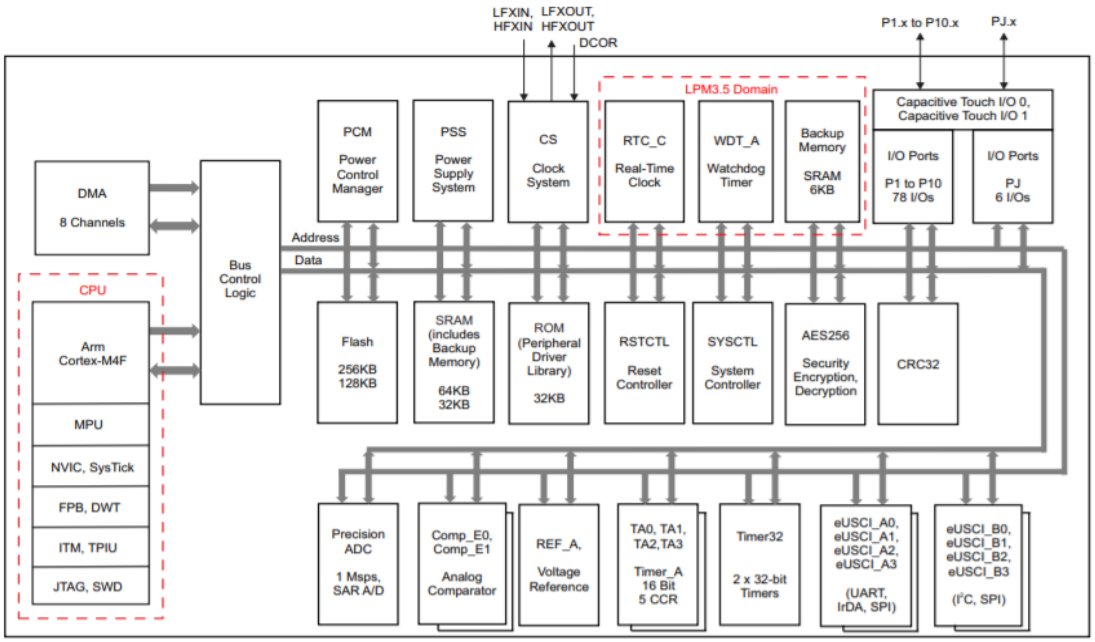


Figura 3. Diagrama de bloques microcontrolador MSP43201R. *[7]*

2.3.2 RSL10

2.3.2.1 Introducción

El RSL 10 es un SoC desarrollado por ON Semiconductor y que incorpora tecnología Bluetooth 5.0. El procesador es un ARM Cortex M3 de 32 bits que permite trabajar en tiempo real y que también incorpora un Nested Vectored Interrupt Controller (NVIC) para la gestión de interrupciones. [8]

2.3.2.2 Características

Las características principales de este microcontrolador son las siguientes:

1. **Procesador:** Es un Cortex ARM M3 de 32 bit que trabaja a una frecuencia de trabajo de 48 MHz. También cuenta con un procesador secundario (LPDSP32) añadido para soportar de manera eficiente las tareas de procesamiento de señal digital

2. **Memoria:** Cuenta con una memoria flash de 384 KB, 24 KB de RAM y 4 KB de ROM.

3. **Relojes y osciladores:** El sistema de relojes tiene como reloj principal SYSCLK el cual puede estar alimentado por diferentes osciladores internos o externos con una frecuencia máxima de trabajo de 48 MHz.

4. **ADC:** Cuatro conversores analógico - digital con una precisión de 8 o 14 bits puede una frecuencia de muestreo de 50*10^3 muestras/segundo, en 8 canales de entrada.

5. **DIO:** El microcontrolador cuenta con 1 puerto en el que hay 16 pines digitales que permiten configurar tanto la corriente máxima de salida como seleccionar la resistencia interna o configurar funciones especiales.

6. **Interfaces de comunicaciones SPI, I2C y UART:** Cuenta con una interfaz UART, una interfaz I2C y dos interfaces SPI para comunicarse con el exterior.

7. **Temporizadores:** 4 temporizadores de propósito general. Estos temporizadores de 24 bits con la capacidad de ser configurable digitalmente a través de preescaladores fijos que van desde 2 hasta 32. Cuenta con tres modos de funcionamiento (“single-shot”, “multiple-shot”, and “free-run”) y tienen una interrupción asociada.

Se pueden apreciar más claramente todos estos periféricos en el diagrama de bloques que proporciona el fabricante como se aprecia en la Figura 4.

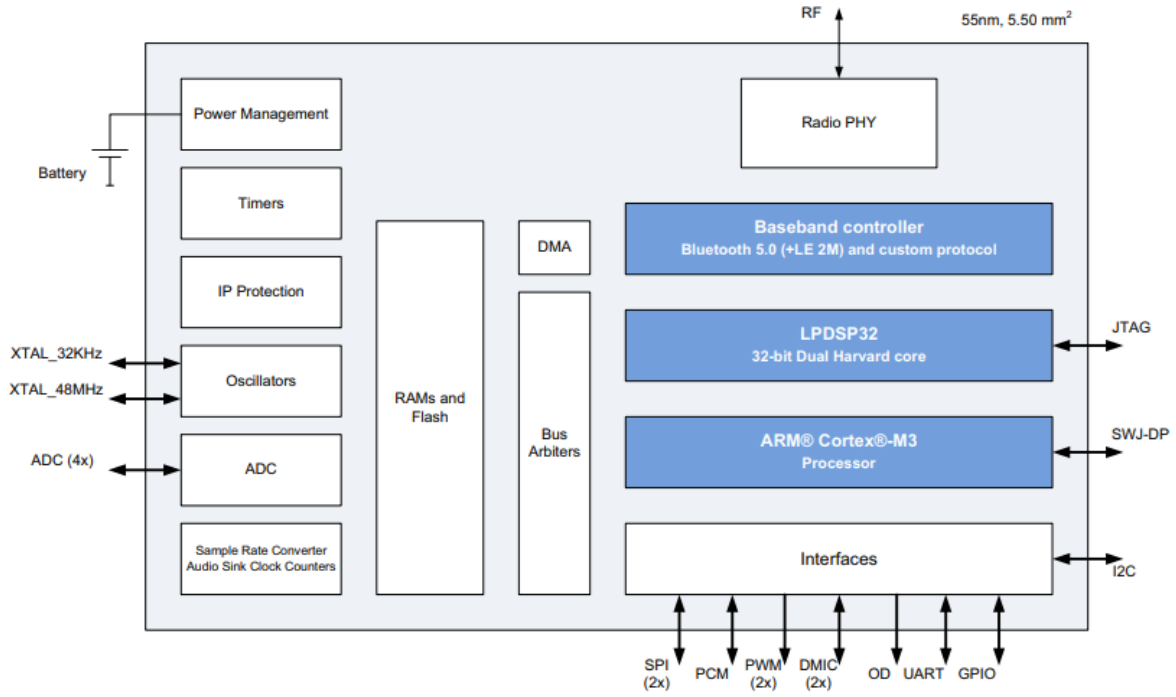


Figura 4. Diagrama de bloques microcontrolador RSL10. Fuente: ON Semiconductor. *[9]*

2.4 Código de estados

Dado que los procesos internos que ejecuta la HAL no tienen por qué ser conocidos por un programador que esté haciendo uso de la HAL, se ha de definir una serie de códigos de estado le permitan estar informando del resultado de las operaciones que se están llevando a cabo. Estos códigos de estado han sido previamente definidos en la capa de Middleware y son notificados al usuario tras finalizar una serie de operaciones que ejecutadas por la HAL.

Todos los códigos han sido numerados y definidos en el archivo Driver_Common.h y contiene tanto los códigos que expresan tanto los mensajes satisfactorios que indican que el periférico ha cumplido correctamente la tarea encomendada (este sería “DRIVER_OK”), hasta aquellos que indican que se ha producido un problema en la capa de usuario como “DRIVER_ERROR_USER_LAYER”.

En la Figura 5 se puede ver la definición y explicación de cada uno de los códigos utilizados, así como el número de error definido.

Número error	Código de Estado	Significado
0	DRIVER_OK	Operación completada con éxito.
1	DRIVER_ERROR	Error genérico
2	DRIVER_ERROR_BUSY	El Driver está realizando una tarea
3	DRIVER_ERROR_TIMEOUT	Se ha excedido el tiempo de realizar una operación
4	DRIVER_ERROR_UNSUPPORTED	Operación no soportada para el microcontrolador
5	DRIVER_ERROR_PARAMETER	Uno de los parámetros pasados es incorrecto
6	DRIVER_ERROR_SPECIFIC	Un error específico de la HAL ha sucedido
7	DRIVER_INITIALIZED	El Driver ha sido inicializado correctamente
8	DRIVER_NOT_INITIALIZED	Error, el Driver no ha sido inicializado
9	DRIVER_CONFIGURED	El Driver ha sido configurado correctamente
10	DRIVER_NOT_CONFIGURED	El Driver no ha sido configurado correctamente
11	DRIVER_ERROR_INCORRECT_INTERFACE_NUMBER	El número de Interfaz pasado por parámetro no es correcto
12	DRIVER_ERROR_FROM_USER_LAYER	Un error ha sucedido en la capa de Usuario

Figura 5. Códigos de estados implementados. [24]

2.5 Herramientas de trabajo

2.5.1 CMSIS

El CMSIS (Cortex Microcontroller Software Interface Standard – Interfaz estándar de software para microcontroladores con procesador ARM Cortex) es una capa de acceso al hardware (HAL) especializada en procesadores ARM Cortex para los que proporciona interfaces de software y otras herramientas de desarrollo. [14]

Este estándar define una serie de componentes y reglas que pueden ser aprovechados por los programadores de microcontroladores facilitándoles las labores de programación.

Estos componentes de CMSIS son las Interfaces para la Programación de Aplicaciones (de sus siglas en inglés API) para los procesadores ARM Cortex M y sus periféricos, las colecciones de bibliotecas para procesadores de señales digitales, las funciones para trabajar con sistemas operativos en tiempo real etc. [14]

De todas estas herramientas, sólo dos han sido utilizadas en el proceso de programación de la HAL, el núcleo del procesador Cortex-M (llamado Núcleo M) y la herramienta SVD.

2.5.1.1- Núcleo M

Este componente de CMSIS define un conjunto de funciones para el procesador Cortex-M y sus periféricos. Contiene información sobre los registros propios de control del Hardware del procesador ARM Cortex M (como los del temporizador SysTick, los de NVIC (Nested Vectored Interupt Table – Tabla de gestión de interrupciones), las funciones propias del procesador y los registros para la gestión de la FPU.

Cuenta también, con los métodos para la inicialización de la CPU (SystemInit()) y un sistema para determinar la frecuencia del reloj de la CPU que configura el Temporizador Systick. [15]

De entre estas herramientas presentes en el Núcleo-M, para la programación de la HAL se han aprovechado estos métodos de inicialización de la CPU y los registros definidos para la gestión por parte del procesador de las interrupciones (NVIC) evitando así tener que desarrollarlos de cero.

En la Figura 6 se puede ver la representación gráfica de los registros de control asociados al periférico NVIC que están declarados en el Núcleo-M.

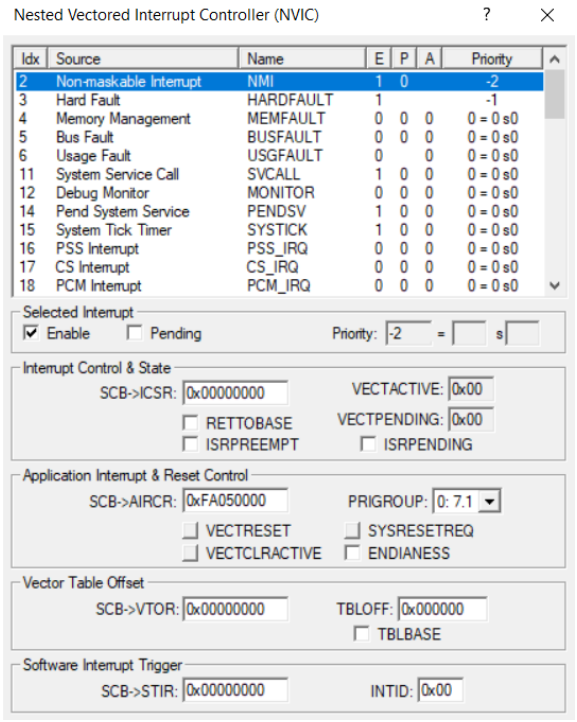


Figura 6. Representación gráfica de los registros del periférico NVIC. *[24]*

2.5.1.2 SVD

SVD es la segunda herramienta utilizada de CMSIS (CMSIS-SVD).
Este componente se encarga de mostrar gráficamente los registros de los periféricos mapeados en memoria, así como formalizar las reglas para que las puedan seguir los fabricantes de los microcontroladores.

Este formato es capaz de representar gráficamente y dar información desde los registros de un periférico hasta la definición y el propósito de un campo de bit (espacio de memoria que ocupa 1 bit de información) de un registro. [16]

SVD ha tenido una gran utilidad para la validación de la HAL con los dos microcontroladores antes mencionados puesto que permite hacer pruebas de configuración de Hardware con el microcontrolador en funcionamiento, evitando así, tener que cargar múltiples veces un mismo código hasta conseguir el funcionamiento deseado.

Un ejemplo de esta descripción gráfica en SVD se puede ver en la Figura 7, donde se muestra la interfaz DIO (pines digitales).
En la figura se puede observar cómo SVD permite conocer el valor actual del registro PAIN (muestra los valores de los pines configurados como entrada de los puertos 1 y 2).

Figura 7. Registros del periférico DIO. *[24]*

2.5.2 Keil MDK

Keil MDK es un entorno de desarrollo de software especializada en dispositivos de microcontroladores con ARM Cortex-M. Este entorno incluye un gran número de herramientas para mejorar las posibilidades de desarrollo como un depurador de código, un compilador para lenguajes C / C ++ o un IDE µVision. [13]

La elección de esta herramienta se debe a su especialización en microcontroladores que cuentan con un ARM Cortex con los beneficios que esto supone, ya que no sólo permite utiliza el estándar CMSIS y por lo tanto se puede hacer uso de los componentes descritos en el capítulo "2.5.1 CMSIS" sino que su compilador también permite ver el estado de las variables informáticas que se hayan definido en cada paso de ejecución del código, se puede depurar el programa paso a paso (esto es poder ejecutar el código desarrollado instrucción a instrucción) lo cual sirve para detectar errores o verificar que el funcionamiento del mismo es el esperado.

Además, este entorno permite llevar a cabo optimizaciones de código y modificar características del proceso de depuración y compilación del código.

Por todas estas características que Keil ha sido el entorno de desarrollo elegido para configurar la HAL.

2.5.3- Doxygen

Doxygen es un generador de documentación de software en el que la documentación se genera automáticamente a partir del código. Doxygen incluye referencias cruzadas entre documentación y código, de modo que el lector puede saltar fácilmente entre el documento y el código [17].

Doxygen es compatible con un gran número de lenguajes de programación como son C, C ++, C #, D, Fortran, IDL, Java, Objective-C, Perl, PHP, Python, Tcl y VHDL, por lo que, para el caso de programación de la HAL, que es C, esta herramienta es válida.

El funcionamiento de Doxygen sólo requiere que el programador incluya un asterisco en los comentarios que tenga su código. Con esto, la herramienta detecta cual es la estructura sobre la que se está haciendo el comentario y genera la documentación en relación a ello.

Un ejemplo de esto se puede ver en la Figura 8. Al incluir el asterisco recuadrado en rojo, Doxygen asocia el comentario "*Defines the posible states of the internal resistor of the channel*" con la enumeración "DIO_RESISTOR_t".

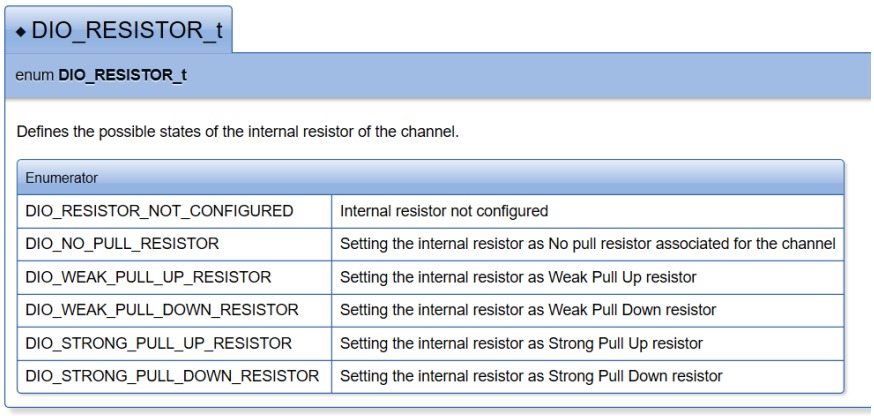
```
/**
* Defines the possible states of the internal resistor of the channel.
*/
typedef enum
{
    DIO_RESISTOR_NOT_CONFIGURED     = 0,  /**< Internal resistor not configured */
    DIO_NO_PULL_RESISTOR            = 1,  /**< Setting the internal resistor as No pull resistor associated for the channel */
    DIO_WEAK_PULL_UP_RESISTOR       = 2,  /**< Setting the internal resistor as Weak Pull Up resistor */
    DIO_WEAK_PULL_DOWN_RESISTOR     = 3,  /**< Setting the internal resistor as Weak Pull Down resistor*/
    DIO_STRONG_PULL_UP_RESISTOR     = 4,  /**< Setting the internal resistor as Strong Pull Up resistor*/
    DIO_STRONG_PULL_DOWN_RESISTOR   = 5,  /**< Setting the internal resistor as Strong Pull Down resistor*/

}DIO_RESISTOR_t;
```

Figura 8. Documentación con Doxygen. *[24]*

Doxygen genera documentación de las estructuras de datos y de los elementos que contiene y solamente hay que incluirles otro asterisco dentro del comentario de cada elemento (como se puede apreciar en todos los comentarios recuadrados en color azul en la enumeración de la Figura 8) para crear la documentación sobre ellos.

Siguiendo estas reglas, Doxygen genera una documentación clara y ordenada en base a los comentarios realizados por el programador como puede verse en la Figura 9 para la enumeración que se ha tomado como ejemplo.



◆ DIO_RESISTOR_t

enum **DIO_RESISTOR_t**

Defines the possible states of the internal resistor of the channel.

Enumerator	
DIO_RESISTOR_NOT_CONFIGURED	Internal resistor not configured
DIO_NO_PULL_RESISTOR	Setting the internal resistor as No pull resistor associated for the channel
DIO_WEAK_PULL_UP_RESISTOR	Setting the internal resistor as Weak Pull Up resistor
DIO_WEAK_PULL_DOWN_RESISTOR	Setting the internal resistor as Weak Pull Down resistor
DIO_STRONG_PULL_UP_RESISTOR	Setting the internal resistor as Strong Pull Up resistor
DIO_STRONG_PULL_DOWN_RESISTOR	Setting the internal resistor as Strong Pull Down resistor

Figura 9. Documentación generada con Doxygen de la enumeración sobre la *Figura 8. [24]*

Con esta idea de generar documentación a partir de los comentarios en el código se consigue tener bien comentado el programa que se esté llevando a cabo y se genera una documentación extra para poder estudiar más cómodamente el software desarrollado.

Esta forma de trabajar fue la que decantó usar Doxygen como generador de documentación para la HAL y se puede consultar los archivos generados tras comentar todo el código en el Anexo III Documentación Doxygen.

2.6 Estructura general HAL

Una HAL asegura que las funciones para el control de una interfaz sean siempre las mismas. Esto se consigue creando dos capas intermedias, la capa de Middleware y la capa Driver. Estas se encuentran entre la capa de Aplicación (donde se desarrolla el código principal) y la capa de Hardware (donde se definen las características del microcontrolador). Ver Figura 10.

Estas capas se relacionan entre sí para conseguir que un usuario que está creando su programa en la capa de aplicación consiga interactuar con todas las capas hasta llegar a la capa de Hardware para crear un archivo ".c" que tras compilarlo se carga en el microcontrolador para que este empiece a ejecutarlo.

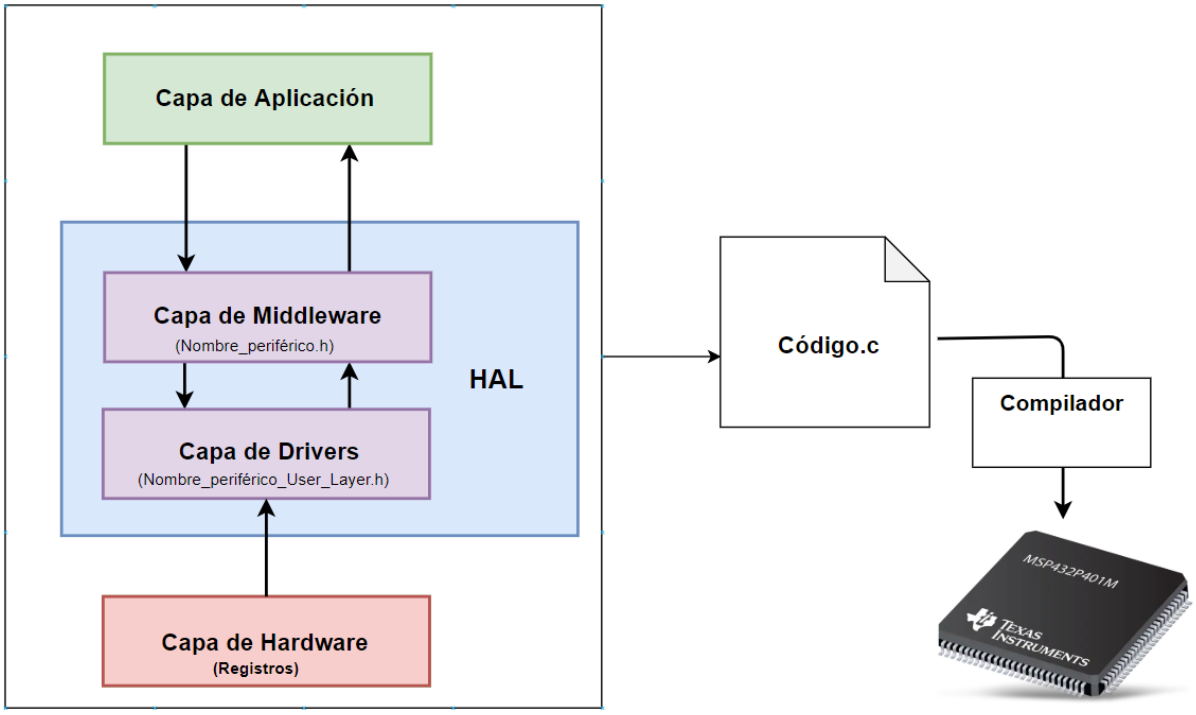


Figura 10. Estructura general de la HAL desarrollada. *[24]*

En la capa de Middleware se encuentran definidas todas las funciones que controlan una interfaz. Esta capa nunca debe ser modificada por el usuario manteniendo con ello el nombre de las funciones que contiene, consiguiendo que para cualquier microcontrolador el nombre de estas sea el mismo.

Además, existe otra ventaja asociada a que esta capa no va a ser modificada nunca y es que se pueden implementar en su interior secuencias de operaciones que son utilizadas frecuentemente por los desarrolladores y son independientes del microcontrolador (como las operaciones que hay que realizar para garantizar el envío de información a través de los buses de comunicación). Al ser implementadas en esta capa, el usuario puede disfrutar de ellas sin tener que volver a reprogramarlas con cada cambio de microcontrolador.

La capa de Driver es la capa en la que se incluyen aquellas funciones a las que la Middleware Layer llama cuando necesita interactuar con el Hardware. Las funciones de esta capa están inicialmente vacías y cuando un programador empieza a trabajar con un nuevo microcontrolador, debe rellenarlas, de tal

forma que interactúen con el Hardware como se indica en los comentarios incluidos dentro de cada una de estas funciones.

Por último, la capa de Hardware es una capa externa a la HAL que debe aportar el usuario. Esta capa contiene los registros y direcciones de memoria para acceder a cada periférico y es normalmente suministrada por los fabricantes de cada microcontrolador.

Con esta estructura se consigue que cuando un usuario decide trabajar con un nuevo microcontrolador, modificando únicamente la capa de Hardware y rellenando la capa de Driver es capaz de trabajar siempre con las mismas funciones y algoritmos típicos sin necesidad de implementarlos de nuevo, con el consiguiente ahorro de tiempo.

2.6.1 Características capa Middleware

La capa de Middleware se ha comentado que nunca va a ser modificada garantizando que nunca cambien los nombres de las funciones en su interior. Estas son sólo algunas de las características, pero realmente la capa de Middleware tiene cuatro propósitos principales:

1- Definir todos los prototipos de funciones de cada interfaz a las que los programadores podrán llamar desde la capa de aplicación para cada uno de los periféricos.

2- Incorporar un sistema de verificación que permita detectar fallos en el paso de parámetros por parte del usuario, permitiendo evitar e informar de posibles errores durante el proceso de configuración del Hardware, desde los pequeños errores en la configuración de las interfaces, como errores con la configuración de los pines de salida, hasta errores más graves como sería configurar el reloj principal del microcontrolador con un oscilador externo que se encontrara deshabilitado.

3- Implementar en las interfaces de comunicaciones los algoritmos de envío y recepción de datos para que no sea necesario añadirlos por parte del usuario y sólo tenga que preocuparse de indicar en qué registros se encuentran los registros de escritura y recepción. Estos algoritmos no dependen del microcontrolador y cada vez que se cambia el microcontrolador seguirán siendo los mismos.
Por supuesto, si el usuario quisiera hacer uso de sus propios algoritmos de envío y recepción de datos podría incorporarlos de forma externa modificando la capa de Middleware o incorporar a la HAL un archivo externo con su implementación.

4- Hacer llamadas a la capa Driver. Puesto que la capa de Middleware no tiene información sobre el microcontrolador necesita ayudarse de la capa de Driver para que sea esta la que interactúe con el microcontrolador y ejecute las operaciones que se le soliciten desde la capa de Middleware.

2.6.2 Características capa Driver

La capa de Driver a diferencia de la capa de Middleware contiene software que ya conoce los aspectos técnicos propios del microcontrolador de trabajo.
En concreto, esta capa hace uso de los registros del microcontrolador definidos en la capa de Hardware. Los registros son direcciones de la memoria del microcontrolador que cuando son modificados desempeñan una serie de acciones sobre el mismo Hardware.

Esta capa inicialmente sólo contiene los prototipos de las funciones a las que la capa de Middleware va a llamar, pero el cuerpo de estas funciones está vacío y el desarrollador que cambia de microcontrolador debe rellenar.
Para completarlas, el usuario cuenta con la ayuda de comentarios detallados escritos dentro de cada una de estas funciones vacías.

Así pues, los objetivos de esta capa son:

1. Definir todos los prototipos de las funciones que el usuario tendrá que completar con los registros del microcontrolador, específicos de cada microcontrolador. Estas funciones incluyen comentarios acerca de lo que el usuario debe rellenar dentro del cuerpo de estas.
   Se han optimizado las funciones de forma que sólo requería por parte del usuario la mínima definición necesaria para que la función opere correctamente, acelerando aún más el trabajo a realizar con cada nuevo microcontrolador.

2. Retornar a la capa de Middleware un código de estado para que desde esta capa se notifique al usuario el resultado de la operación realizada.

2.6.3 Interfaces en la HAL

Las interfaces de la HAL implementada han sido diseñadas para trabajar con los periféricos más comunes con los que cuentan los microcontroladores, los de comunicación serie (SPI, I2C y UART), Temporizadores, ADC y pines digitales de Entrada/Salida a través de una serie de interfaces que se encuentran dentro de la capa de Middleware y la capa de Driver. Véase la Figura 11.

La capa de Middleware se ha decidido llamar también como capa del creador (Creator_Layer) puesto que aquí se declaran todas las enumeraciones y macros que entienden las funciones de cada interfaz y no debería ser modificadas por el usuario, así que como éste no debería interactuar con ellas.
En cambio, la capa de Drivers también se llama capa de usuario (User_Layer) para indicar que esta capa es la que puede modificar. De esta forma, para el programador es sencillo saber si un archivo es de una capa u otra, puesto sólo deben ser modificados aquellos en los que el nombre de cada interfaz termina en "xxx_User_Layer".

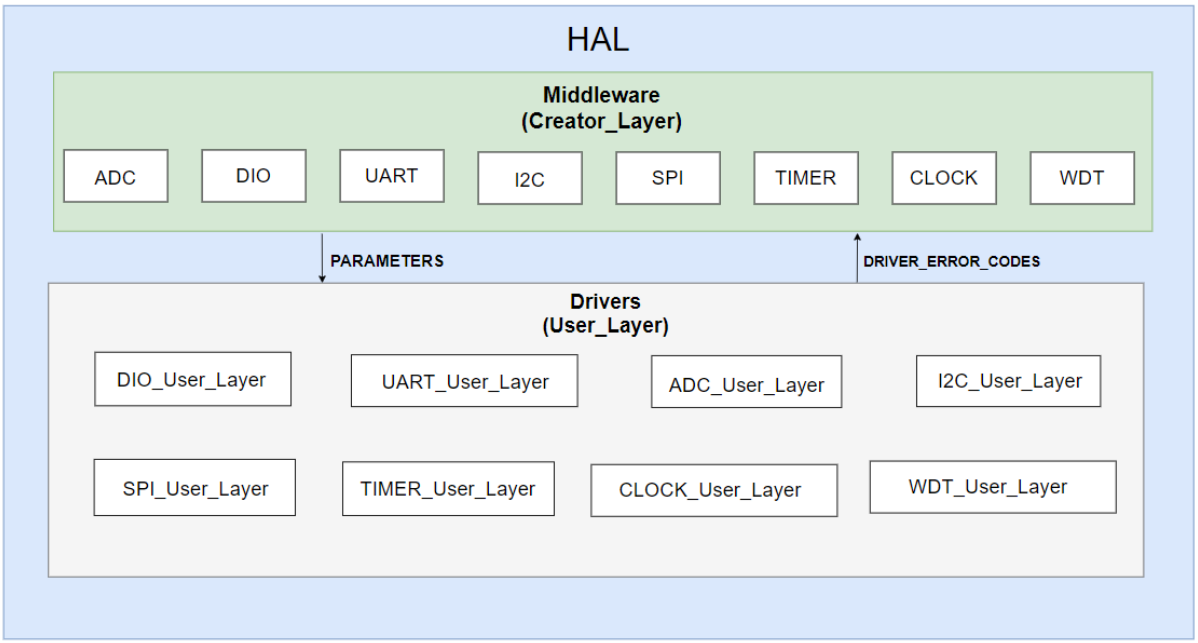


Figura 11. Estructura interna de la Middleware y de la capa de Drivers. *[24]*

Dentro de la capa de Middleware se incorporan archivos ".h" y ".c" que son los que contienen las funciones de control de cada periférico. Como un microcontrolador puede tener varios periféricos iguales; por ejemplo, dos temporizadores, las interfaces desarrolladas deberán ser capaces de controlar todos y cada uno de los periféricos de un mismo tipo; esto es, si tengo tres interfaces de comunicación serie UART, mi interfaz debe ser capaz de interactuar con las tres, haciendo uso de las mismas funciones, sin tener que introducir nuevo código.

2.6.4 Metodología para el desarrollo de una interfaz

El proceso de configuración que se ha seguido para el desarrollo de una interfaz ha sido el siguiente:

1. Búsqueda de información sobre las características comunes de un mismo periférico de varios fabricantes diferentes para determinar qué funcionalidades son las que más se repiten y debe implementar nuestra interfaz.

2. Se definen las estructuras de datos que van a ser necesarios para modelar cada periférico, añadiendo comentarios que clarifican el significado de estos.

3. Se diseña una tabla (a nivel de software) que recogerá la información de cada tipo de periférico (ADC, SPI, UART etc.) durante la ejecución del código. Esta tabla contendrá en todo momento información referente a el estado del periférico y le servirá a la HAL como medio de comprobar el estado de cada periférico, así como evitar que el desarrollador cometa errores en la configuración de las interfaces.

4. Se define todo el conjunto de funciones que van a constituir la interfaz.

5. Se implementa el sistema de verificación de parámetros y de devolución de los códigos de estado dentro de cada función.

6. Se incorporan las llamadas desde el cuerpo de las funciones de la capa de Middleware a las de Driver. Para realizar el paso de parámetros cuando se llamen a estas funciones, se ha decidido utilizar una técnica muy usada en la informática que consiste en pasar los parámetros por referencia (mediante un puntero a una struct del mismo tipo que de la que estaba formada la tabla de información del periférico) en lugar de por valor. Con esta técnica se consigue evitar excesos de uso de memoria que pueden ser innecesarios.

7. Se completan en la capa de Drivers las funciones prototipo a cumplimentar por el usuario. Las funciones tendrán que llevar a cabo las acciones solicitadas dependiendo del parámetro pasado por referencia que se acaba de recibir desde la capa de middleware. También se reportará un código de estado.

8. Por último, se analizará el código de estado, si este fuera DRIVER_OK se actualizará la tabla de información del periférico con el nuevo estado. En caso contrario, el código de estado nos indicará dónde se ha producido el error.

2.6.5 Reglas seguidas para programación de la HAL

Para configurar la HAL se han establecido una serie de reglas a cumplir necesariamente para concluir de forma satisfactoria este proyecto:

1. Sólo los archivos de nombre "*nombre_del_periférico*_User_layer" definidos en la capa de Driver pueden contener información referente a un microcontrolador concreto con el que se está trabajando (registros, direcciones de memoria, funciones de cada pin etc.).

   Por ejemplo, para el manejo del periférico de los pines digitales de entrada y salida, el usuario incluirá en la capa de aplicación una directiva de preprocesador para que incluya la capa dio.h (que se encuentra en la capa de Middleware) que contendrá todas las funciones prototipo definidas que son siempre las mismas.
   A través de las funciones que hay en Dio (en la Middleware), sólo se realizarán llamadas a las funciones en la interfaz inmediatamente inferior, Dio_User_Layer, definida en la capa de Driver por el encargado de adaptar la HAL al nuevo microcontrolador. De modo que sólo la interfaz Dio_User_Layer está incorporando información específica de un microcontrolador concreto. Véase Figura 12.

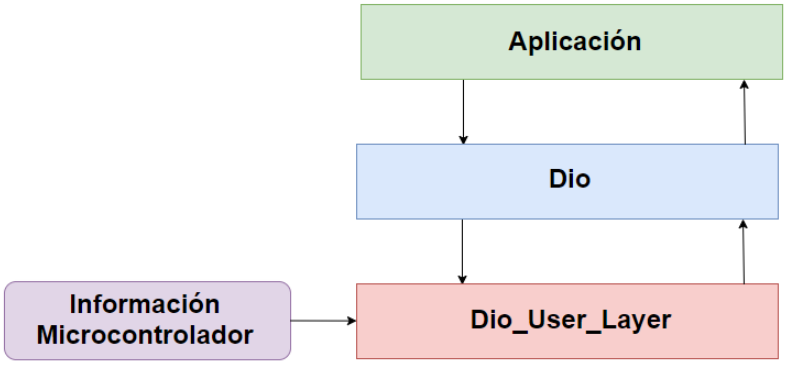


Figura 12. Relación entre capas con la información del microcontrolador *[24]*

2. La implementación de la HAL debe conseguir que el usuario únicamente tenga que llevar a cabo las siguientes tareas:

   - Configurar todos los archivos "xxx_User_Layer". Véase los archivos dentro de la carpeta User_Layer.c que deberán ser completados en la Figura 13.Figura 13.

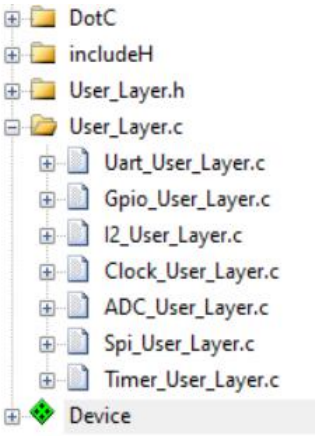


Figura 13.Archivos xxx_User_Layer.h a rellenar por el usuario. [24]

- Completar el archivo llamado Vital_Information.h con la información de su microcontrolador. Véase *Figura 14*.

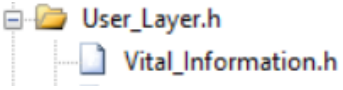


Figura 14. Archivo Vital_Information.h en User_Layer.h. [24]

3. Incluir en las funciones asociadas a los periféricos de comunicaciones que se ejecutan en un microcontrolador cada vez que se produce una interrupción (Handlers), una llamada a las funciones definidas previamente por el creador de la HAL (cuyo nombre se puede encontrar en la información extra de la HAL), para que estas continúen la transmisión/recepción de información. Este tema será tratado con mayor profundidad en el capítulo "2.6.6- Gestión de interrupciones en la HAL".

4. Para cada tipo de periférico, se diseñará una tabla software de información sobre cada una de las interfaces. De esta forma se podrá conocer en cada momento el estado del periférico, permitiendo comprobar que el funcionamiento es el adecuado y evitando posibles errores de configuración.

   Para modelar a nivel de software esta tabla, se ha decidido hacer uso de una estructura de información muy utilizada en informática, llamada vector.
   Cada uno de estos vectores que contendrá la información de una interfaz tendrá el siguiente nombre: *****nombre_del_periférico*_RESOURCES.
   Dado que a priori no se conoce el número de elementos que debe tener cada una de estas tablas, ya que esto depende del microcontrolador que se esté usando en ese momento, será necesario destinar una directiva de preprocesador con el siguiente nombre *nombre_del_periférico*_NUMBER_OF_INTERFACES para conocer este número.

   Así el usuario sólo tendrá que declarar el número de interfaces que tiene el microcontrolador gracias a una directiva de preprocesador que se encuentra en el archivo Vital_Information.h, y la HAL creará la tabla de información con el número de elementos especificados.

Por ejemplo, si en el archivo Vital_Information.h a la directiva UART_NUMBER_OF INTERFACES se le añade el número 3, la HAL creará una tabla llamada UART_RESOURCES con tres elementos como se puede observar en la Figura 15.

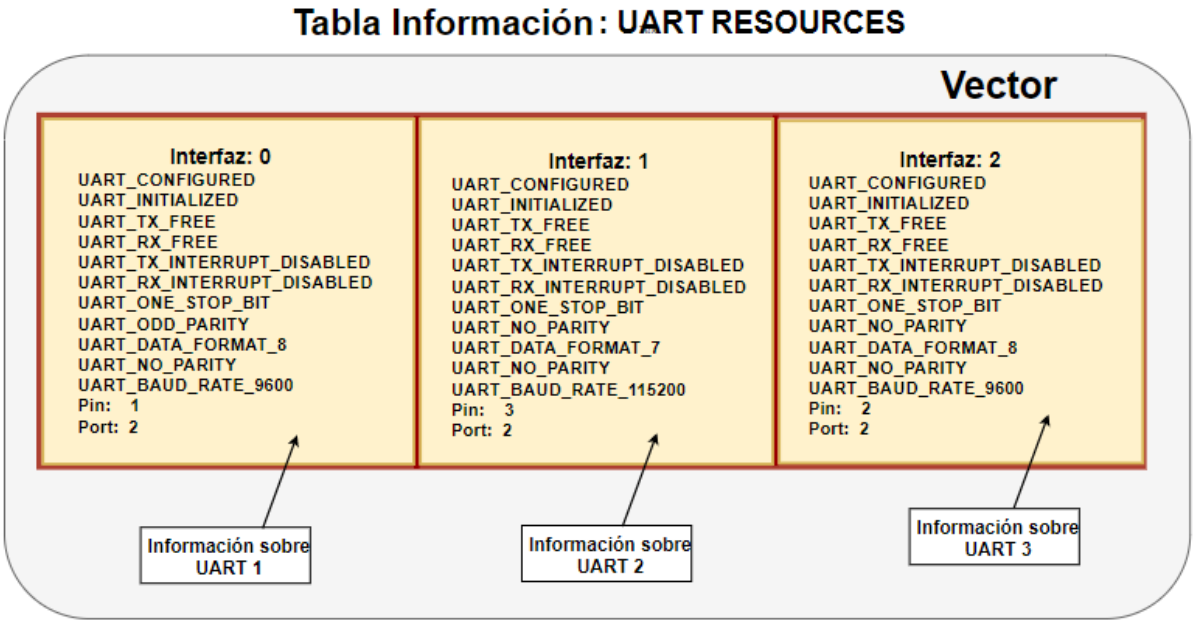


Figura 15. Tabla de información para la interfaz de UART. *[24]*

Con cada tabla se tiene la información en tiempo real del estado de la interfaz del microcontrolador a la que haga referencia.

5. Se deben implementar códigos de estado que permitan al usuario verificar durante cada operación como se ha llevado a cabo.

6. Incorporar un sistema de verificación de parámetros para evitar fallos en el uso de las funciones por parte del usuario.

2.6.6- Gestión de interrupciones en la HAL

Una interrupción es un evento producido por un elemento interno o externo al microcontrolador que le indica a este que debe realizar las acciones que previamente hubiera establecido el programador, si esta interrupción se encontraba habilitada.

La gestión de las interrupciones es un punto realmente importante a la hora de crear una HAL ya que son muy utilizadas y proporcionan gran valor agregado a los microcontroladores.

Es importante destacar que cada interrupción es atendida por unas funciones especiales, propias de cada periférico, llamadas Handlers. Estos Handlers son definidos en un lenguaje llamado ensamblador y en él los desarrolladores pueden configurar los nombres de cada uno de ellos.

El que los nombres de los Handlers sean modificables y establecidos por el usuario supone un problema a la hora de implementar la HAL puesto que es imposible para el creador de esta saber el nombre que les será dado a cada uno de ellos a priori, impidiéndole a la HAL no hacer uso de interrupciones. Por lo

tanto, hay que diseñar una solución que evite este problema.

2.6.6.1 Estrategia de actuación con las interrupciones

La decisión adoptada respecto a la gestión de las interrupciones ha sido tomada en base a lo que se han considerado dos tipos diferentes de interrupciones: las interrupciones que tras terminar las instrucciones de su Handler dejan un proceso sin terminar en la HAL (generadas por los periféricos de comunicaciones), llamadas interrupciones A y las interrupciones que cuando finaliza la última instrucción definida en su Handler, terminan su intervención (ocasionadas por pines digitales, temporizadores o el ADC), llamadas interrupciones B

Las interrupciones A requieren de la HAL para que continúe con el envío/recepción de información.
Por ejemplo, si se solicita a la HAL enviar una cadena de caracteres, la HAL debe ser la encargada de llevar la cuenta del número de estos caracteres que ya han sido enviados y cuantos faltan por enviar, debe almacenar la cadena de caracteres y también guardar la función que el desarrollador desea que se ejecute una vez finalizado el envío; es decir, tras saltar la interrupción de que el primer carácter ha sido enviado y termine de ejecutar el Handler asociado a este evento, la HAL deberá saber que el proceso aún no ha terminado hasta que se finalice el envío de todos los caracteres, guardando información durante el proceso.

En cambio, las interrupciones B no requieren de la HAL ya que no dejan ningún proceso pendiente.
Por ejemplo, si se habilita la interrupción de un pin digital, para que cuando mida un cambio de voltaje del exterior al microcontrolador, ejecute las instrucciones definidas en el Handler, lo habitual es que estas instrucciones lleven a cabo una acción que no requieren que la HAL almacene información, como puede ser encender un led, activar un temporizador, iniciar una comunicación serie o hacer una medida con el ADC. Estas tareas se hacen en el mismo Handler y la HAL no tiene que almacenar información.

Teniendo clara esta diferencia se ha procedido a actuar de distinta forma para cada tipo de interrupción. Para las interrupciones tipo A, se ha determinado que el usuario incluya en los Handlers ya definidos en la capa de Driver, propios de su microcontrolador, una llamada a una función de la capa de Middleware. Esta función continuará con la transmisión o recepción de datos.

Un ejemplo de cómo llevar a cabo esta práctica con la interfaz de UART se muestra en la Figura 16. En la capa de Driver el usuario deberá incluir en el Handler de la función de interrupción asociada a la UART (recuadrada en color azul) una llamada a la función creada en la HAL (definida en la capa de Middleware) que continua la transmisión de datos (recuadrada en color rojo) como se ve en la Figura 16. A esta función definida en la capa de Middleware hay que pasarle por parámetro el número de la interfaz UART que haya generado la interrupción. En este ejemplo de las doce posibles UART con las que cuenta el microcontrolador, al pasarle por parámetro el "0" indicamos que la interfaz que debe seguir enviando/recibiendo es la número cero.

```
void EUSCIA2_IRQHandler(void){

    /*
    Se comprueba si el flag que se ha activado es el de
    la recepción.
    */
    if((EUSCI_A2->IFG & EUSCI_A_IFG_RXIFG) != 0)
    {
        //Se baja el flag de recepción.
        EUSCI_A2->IFG &= ~EUSCI_A_IE_RXIE;

        // SE LLAMA A LA INTERRUCPCIÓN DEFINIDA EN
        // MIDDLEWARE LAYER pasando el número de interfaz 0
        Uart_Rx_Interrupt_Handler(0);
    }
```

Figura 16. Funcionamiento de un Handler implementado por el usuario. *[24]*

En la Figura 17 se puede ver un ejemplo más general lo que sucede en la HAL con una interrupción tipo A, como puede ser la de la comunicación UART. En este caso el usuario define el Handler asociado a la UART en la capa de Driver llamado en la Figura 16 como "User Interrupt Handler" y desde ahí llama a la función definida en la capa de Middleware llamada "HAL Interrupt Handler", pasándole como parámetro el número de la interfaz UART que ha generado la interrupción, para que esta continúe con la recepción de información.

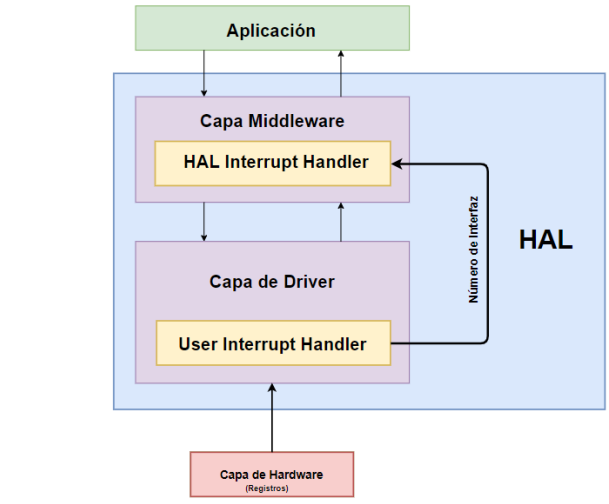


Figura 17. Gestión de la interrupción tipo A por la HAL. *[24]*

Para las interrupciones tipo B, se ha decidido pedir al usuario que cuando quiera utilizar interrupciones asociadas a los periféricos ADC, Temporizador o pines digitales, incorpore en la capa de Aplicación directamente los Handlers que ha definido, evitando así tener que trabajar con los nombres de funciones que se hubieran creado en la HAL.

2.6.7 Verificación Funcional

Para evitar errores en la implementación de la HAL se ha llevado a cabo una estrategia de detección y corrección de errores, que nos permite anticiparnos a errores y fallos de otra índole.

Los pasos seguidos para llevar a cabo esta verificación han sido los siguientes:

1. Elaborar una lista de todos los posibles parámetros que han sido diseñados para ser pasados a las funciones de middleware.
   Véase el siguiente ejemplo con la función Uart_Configure y el parámetro _parity:
   La interfaz de UART tiene una función llamada Uart_Configure y precisa para su correcto funcionamiento de una serie de parámetros que se pueden observar en la Figura 18

```
Uart_Configure(
                const INTERFACE_NUMBER_t   _interface_number,
                const RX_PIN_t             _rx_pin,
                const uint8_t              _rx_port,
                const TX_PIN_t             _tx_pin,
                const uint8_t              _tx_port,
                const uint32_t             _baudios,
                const UART_STOP_BITS_t     _stop_bits,
                const uint8_t              _data_bits_number,
                const uint8_t              _msb_first,
                const UART_PARITY_t        _parity
                );
```

Figura 18. Función Uart_Configuration y sus parámetros. *[24]*

Uno de estos parámetros es "_parity" que es de tipo de dato "UART_PARITY_t" y puede tomar los siguientes valores que se ven en la Figura 19.

```
/**
*     Defines the PARITY of the UART interface
*/
typedef enum
{
      UART_PARITY_NOT_CONFIGURED = 0U,    /**< Uart parity not configured*/
      UART_NO_PARITY             = 1U,    /**< Uart no parity */
      UART_ODD_PARITY            = 2U,    /**< Uart odd parity */
      UART_EVEN_PARITY           = 3U,    /**< Uart even parity*/

} UART_PARITY_t;
```

Figura 19. Identificadores de la enumeración UART_PARITY_t. *[24]*

Ahora, se elaborará una lista de los posibles valores que pueden tener cada uno de los parámetros. Para el caso de "_parity" estos pueden ser los siguientes: UART_PARITY_NOT_CONFIGURED, UART_NO_PARITY, UART_ODD_PARITY y UART_EVEN_PARITY.

2- Se carga en un microcontrolador un código que contenga la función a comprobar con cada una de las opciones que presenta cada parámetro y han sido incluidos en la lista elaborada en el paso 1. Una vez subido el programa al microcontrolador, este será depurado instrucción a instrucción, comprobando en cada momento que el funcionamiento se corresponde con lo esperado para ese parámetro; es decir, se ha configurado el hardware según el parámetro, se ha actualizado la tabla de información del periférico y se han obtenido los códigos de estado.

3- Una vez comprobados todos los parámetros de forma individual, se procede a comprobar su correcto funcionamiento con un segundo microcontrolador, con el objeto de disminuir la probabilidad de que ciertos parámetros no estuvieran suficientemente acotados y pudiesen fallar en otros microcontroladores.

4- En caso de detectar errores se repetirán los pasos 2 y 3 hasta eliminarlos y conseguir el correcto funcionamiento de cada función.

5- Si la función ha operado correctamente, se elaborará documentación sobre ella para dar soporte y facilitar su manejo, dando por concluido con éxito, el proceso de programación de dicha función.

Capítulo 3: Interfaz Gráfica

Una interfaz gráfica de usuario, conocida también como GUI (del inglés Graphical User Interface), es un programa informático que interactúa con el usuario, haciendo uso de un conjunto de imágenes y objetos gráficos para representar la información y las acciones posibles a realizar. [10]

La presente interfaz gráfica ha sido desarrollada con la finalidad de facilitar al usuario el conocimiento de los nombres de las funciones y sus parámetros definidos en la HAL.
Se ha utilizado una librería estándar de Python llamada Tkinter[26] la cual cuenta con un gran número de clases y procedimientos que permiten incluir de forma rápida ventanas, botones, etiquetas y demás recursos gráficos que se usan habitualmente en las Interfaces gráficas (canvas, cursores, deslizadores etc.).

Python es un lenguaje de programación interpretado, cuya filosofía hace hincapié en la legibilidad de su código [11]. El principal motivo por el que se ha utilizado Python en el desarrollo de esta interfaz es debido al amplio uso que se le está dando por parte de la comunidad científica, que cuenta con una gran comunidad, con muchos módulos y librerías disponibles que dan soporte al rápido desarrollo de aplicaciones.

3.1 Principio de funcionamiento

El propósito de la Interfaz Gráfica implementada es ayudar al usuario a conocer las funciones de configuración de periféricos que están disponibles en la HAL, así como las enumeraciones y macros que pueden actuar como parámetros en las funciones de la HAL.

La interfaz desarrollada no es estática; es decir, es capaz de adaptar el número de elementos gráficos que muestra según el microcontrolador que se está usando en ese momento. Para conseguir este efecto se requiere de una serie de archivos que debe completar el programador como se verá posteriormente en el capítulo “3.2 Funcionamiento Interno”.

Cuando la interfaz gráfica se despliega, esta presenta al usuario una serie de botones para seleccionar el periférico que se quiere configurar. Ver Figura 20.

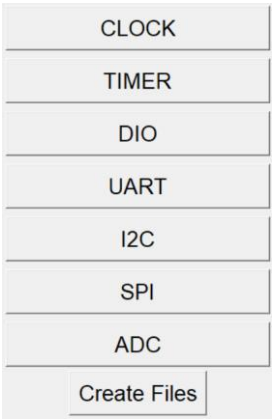


Figura 20. Pantalla principal de selección de periférico a configurar en la Interfaz Gráfica. [24]

Una vez seleccionado el periférico a utilizar, la Interfaz Gráfica muestra un número de recuadros igual al número de periféricos de ese tipo, que tiene el microcontrolador.
Por ejemplo, si el usuario está trabajando con un microcontrolador que cuenta con dos interfaces SPI, la interfaz gráfica sólo mostrará 2 recuadros (interfaz SPI 0, interfaz SPI 1). Ver Figura 21.

Figura 21. Vista interior del periférico SPI en la interfaz gráfica con 2 interfaces.
[24]

Mientras que, en el supuesto de haber definido cuatro interfaces SPI, estas se mostrarán como se aprecia en la Figura 22.

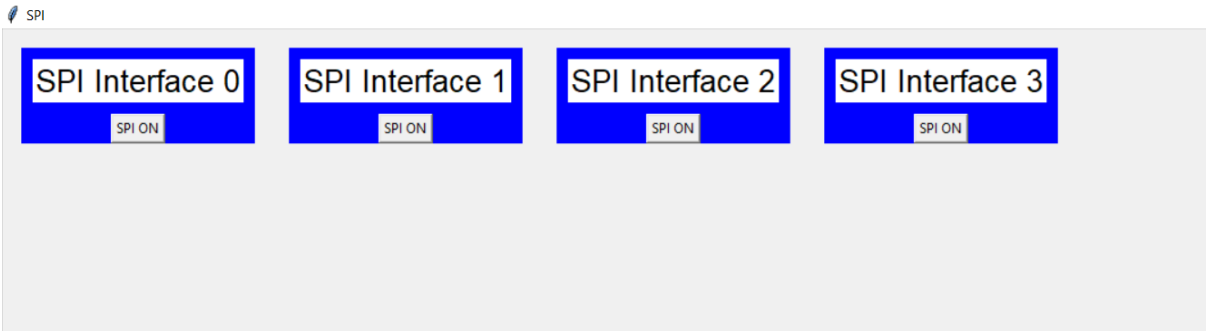


Figura 22. Vista interior del periférico SPI en la interfaz gráfica con 4 interfaces.
[24]

Dentro estos recuadros aparecen un botón que le permiten al programador decidir si quiere o no quiere configurar el periférico correspondiente. (Véase un ejemplo de esto con el periférico DIO en la Figura 23).

Figura 23. Vista del interior de la Interfaz Gráfica en el periférico DIO. *[24]*

Una vez seleccionado el periférico que se desea configurar, haciendo un click sobre el botón de ON, se desplegarán todas las opciones asociadas a la configuración de este periférico. Las selecciones sobre las configuraciones que se van a realizar en este momento se convertirán en los parámetros que serán pasados a la configuración del mismo.
(Ver Figura 24, ejemplo de las opciones cuando se despliegan los periféricos DIO pin 1.1 y pin 1.2)

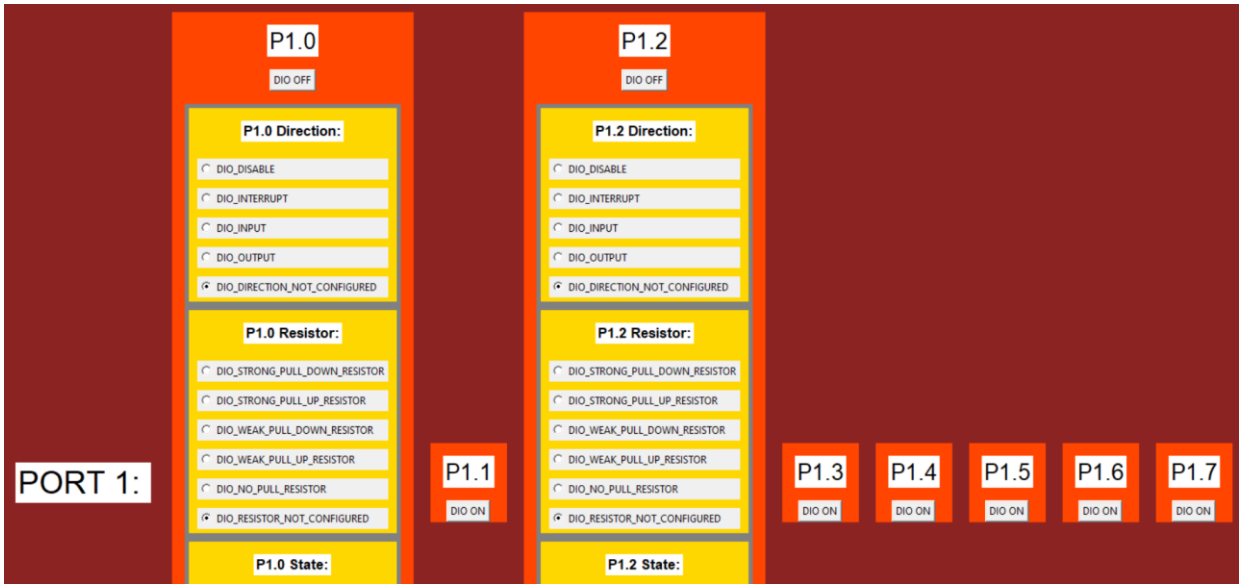


Figura 24. Vista del interior del periférico DIO con los pines P1.0 y P1.2 habilitados. *[24]*

Una vez seleccionadas todas las opciones para cada una de las interfaces que se deseen utilizar y para cada periférico, habrá que hacer un doble “click” al botón de “Create Files” que se muestra en la Figura 20.

Al pulsar este botón, se abrirá un gestor de carpetas indicándonos la ruta donde se desea guardar el archivo que va a generar la interfaz.

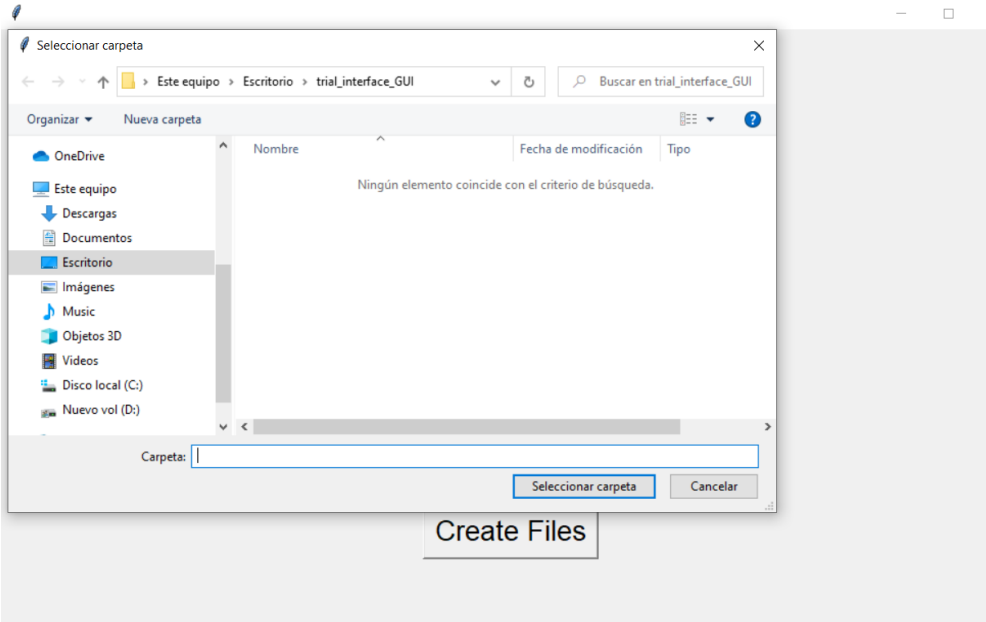


Figura 25. Vista del gestor de archivos que indica donde generar el archivo deseado. *[24]*

Una vez seleccionada la carpeta en la que se guardará el archivo, este se creará en la ruta indicada, a la que se podrá ir y visualizar. Véase Figura 26.

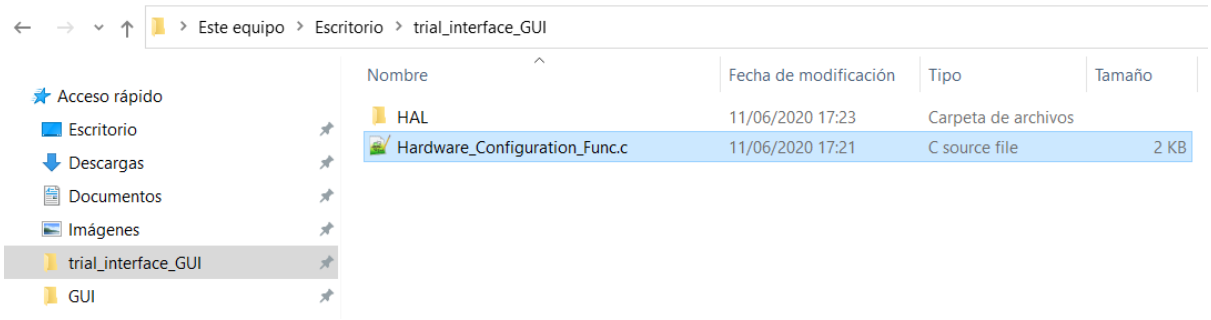


Figura 26. Vista del archivo ".c" generado por la Interfaz Gráfica *[24]*

Dentro de este archivo que se acaba de generar se podrá comprobar que la interfaz gráfica ha incluido, aquellas librerías de la HAL necesarias para el uso de los periféricos elegidos previamente (Figura 27 recuadrado en azul), habrá declarado las funciones de configuración de los periféricos seleccionados (Figura 27 recuadrado en rojo) y habrá generado unos comentarios en el código que informarán sobre la configuración elegida para ese periférico (Figura 27 recuadrado en amarillo).

```
#include "DIO.h"

int main(void)
{

/**********************************************************************************************************************
*************************************************** DIO  CONFIGURATION  ***********************************************
**********************************************************************************************************************/

/*Configurando pin:     P1.0
  Resistencia interna:  DIO_RESISTOR_NOT_CONFIGURED
  Direccion del pin:    DIO_DIRECTION_NOT_CONFIGURED
  Estado inicial:       DIO_STATE_NOT_CONFIGURED
  Drive:                DIO_DRIVE_NOT_CONFIGURED
*/
   Dio_Configuration( 1, 0, DIO_RESISTOR_NOT_CONFIGURED, DIO_DIRECTION_NOT_CONFIGURED, DIO_STATE_NOT_CONFIGURED, DIO_DRIVE_NOT_CONFIGURED );

/*Configurando pin:     P1.2
  Resistencia interna:  DIO_RESISTOR_NOT_CONFIGURED
  Direccion del pin:    DIO_DIRECTION_NOT_CONFIGURED
  Estado inicial:       DIO_STATE_NOT_CONFIGURED
  Drive:                DIO_DRIVE_NOT_CONFIGURED
*/
   Dio_Configuration( 1, 2, DIO_RESISTOR_NOT_CONFIGURED, DIO_DIRECTION_NOT_CONFIGURED, DIO_STATE_NOT_CONFIGURED, DIO_DRIVE_NOT_CONFIGURED );


}
```

Figura 27. *Vista del interior del archivo generado por la Interfaz Gráfica.* [24]

Resaltar que aquellas opciones que el usuario había elegido previamente de forma gráfica se han convertido en los parámetros de estas funciones de configuración.

Un problema de las HAL es que hay que conocer todas las enumeraciones y macros que ha definido su creador para hacerla funcionar correctamente, este problema se soluciona con la Interfaz gráfica, porque es ella misma la que, en base a las selecciones realizadas previamente, sabe cuáles son las enumeraciones que deben ser utilizadas.

3.2 Funcionamiento Interno

Como se ha comentado previamente, la Interfaz Gráfica es capaz de adaptar los elementos gráficos que le muestra al usuario según el microcontrolador con el que esté trabajando en ese momento, esto se consigue mediante un sistema de lectura de archivos Json y Python.

3.2.1 JSON

JSON (JavaScript Object Notation - Notación de Objetos de JavaScript) establece un formato ligero basado en el Lenguaje de Programación JavaScript, que define una forma de organizar datos y es utilizado para llevar a cabo un alto volumen de intercambio de datos [12].

El formato Json cuenta con una serie de ventajas:

1. Es muy utilizado actualmente para organizar y clasificar información, lo que permite a los usuarios que decidan utilizar la interfaz gráfica, encontrar mucha información referente al mismo para aprender a manejarlo.

2. Existen multitud de recursos para Python capaces de extraer la información contenida en los archivos Json, facilitando las labores de la programación de la actual Interfaz Gráfica y de las futuras versiones.

3. La información contenida en un archivo Json está organizada de forma visual y legible, permitiendo a golpe de vista, tener una visión global de la información existente en el interior del documento y como ésta se encuentra estructurada. Ver Figura 28.

Estás características que ofrece Json frente a otros formatos de clasificación de información, como podría ser XML, también muy popular y muy utilizado para el desarrollo web, han hecho que el formato Json sea el formato elegido para el desarrollo del proyecto.

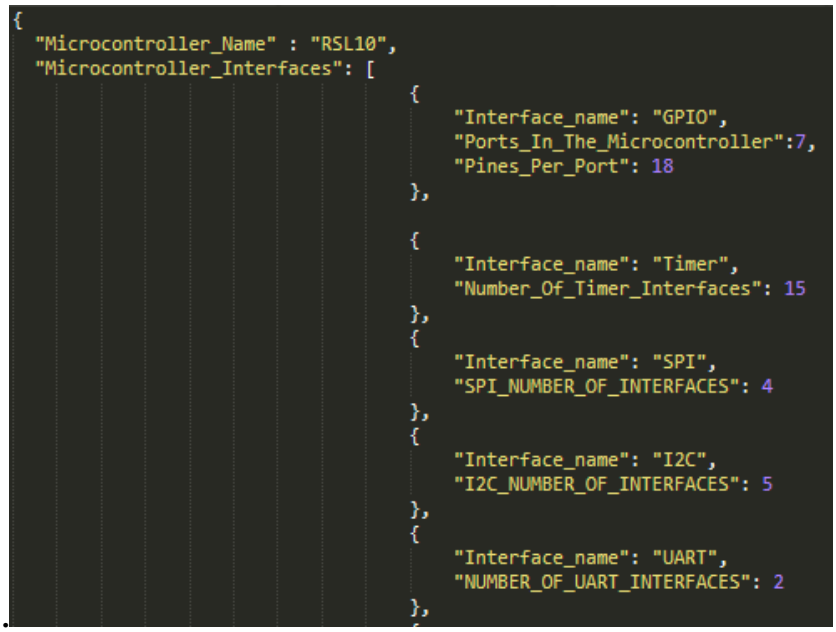


Figura 28. Vista interior del archivo de configuración del microcontrolador. *[24]*

3.2.2 Procedimiento de adaptación de la Interfaz Gráfica

El proceso para crear una Interfaz gráfica adaptable al microcontrolador consiste en que previo a generar todos los recursos gráficos mostrados al usuario, se debe recoger la información referente al microcontrolador que se esté usando en ese momento y también referente a características de la HAL.

Toda esta información se encontrará en dos archivos Json y deberán ser completados uno por el creador de la HAL y otro por el progrmador del microcontrolador. Estos archivos son llamados "Creator_Template.json" y "User_Template.json". Gracias a estos dos archivos, la Interfaz Gráfica puede generar todos los elementos gráficos que sean necesarios. El contenido de estos archivos se analizará en los próximos capítulos "3.2.2.1 Creator_Template.json" y "3.2.2.2 User_Template.json"

Posteriormente, el desarrollador configura gráficamente cada periférico como se ha visto en el punto"3.1 Principio de funcionamiento". Tras esta interacción, la Interfaz Gráfica genera un archivo de salida como el visto en la Figura 27.

El modo de funcionamiento general de la Interfaz gráfica se puede observar en la Figura 29.

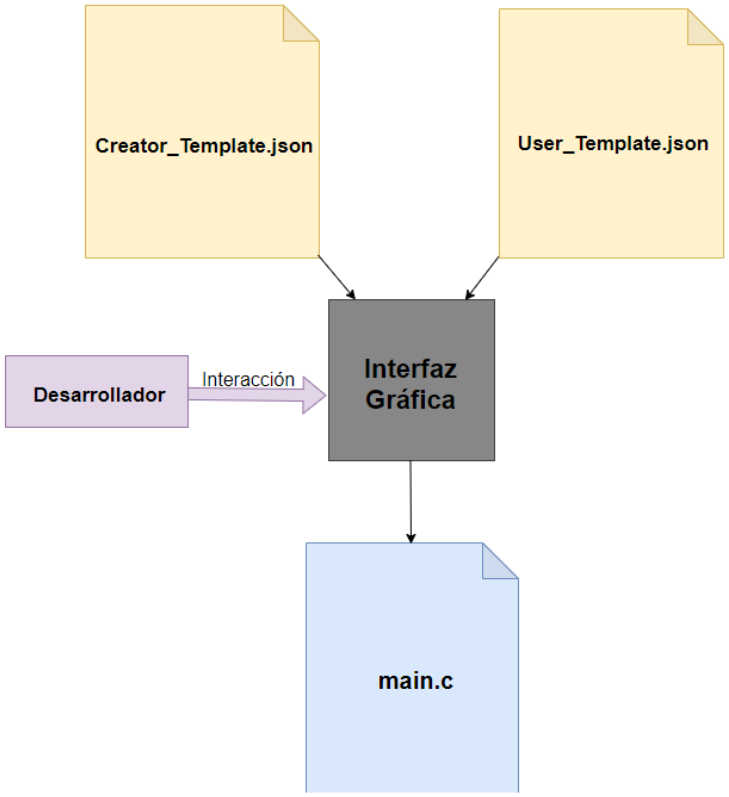


Figura 29. Sistema de funcionamiento de la Interfaz Gráfica. *[24]*

3.2.2.1 Creator_Template.json

El archivo “Creator_Template.json” debe ser completado por el creador de la HAL y en él se deben incluir todas las enumeraciones y macros que han sido definidas en la HAL y el orden de los parámetros de cada función.
Este archivo es el que proporciona a la Interfaz Gráfica toda la información referente de las características de la HAL y será en base al contenido de este archivo por el que se permitirá al usuario elegir entre una serie de opciones de configuración cuando selecciona un periférico. Ver Figura 30.

Figura 30. Vista interior del Archivo Creator_Template.json *[24]*

Como se puede ver si se comparan las Figuras 24 y 30, se aprecia como todos los elementos dentro de “DIO_RESISTOR_t” (DIO_NO_PULL, DIO_WEAK_PULL_UP_RESISTOR etc.) son los que aparecerán en la interfaz gráfica para que el usuario pueda seleccionarlos gráficamente.

Toda la información del archivo Creator_Template.json se extrae directamente del archivo dio.h de la HAL como se ve en la Figura 31.

```
/**
* Defines the possible states of the internal resistor of the channel.
*/
typedef enum
{
        DIO_RESISTOR_NOT_CONFIGURED     = 0,  /**< Resistor not configured */
        DIO_NO_PULL_RESISTOR            = 1,  /**< No Pull resistor*/
        DIO_WEAK_PULL_UP_RESISTOR       = 2,  /**< Weak Pull Up resistor */
        DIO_WEAK_PULL_DOWN_RESISTOR     = 3,  /**< Weak Pull Down resistor*/
        DIO_STRONG_PULL_UP_RESISTOR     = 4,  /**< Strong Pull Up resistor*/
        DIO_STRONG_PULL_DOWN_RESISTOR   = 5,  /**< Strong Pull Down resistor*/
} DIO_RESISTOR_t;
```

Figura 31. Vista interior del Archivo Creator_Template.json *[24]*

3.2.2.2 User_Template.json

El archivo User_Template.json" se incluyen las características del microcontrolador con el que se esté trabajando y debe ser completado por el programador del microcontrolador.
Inicialmente este archivo se comporta como una plantilla que el usuario debe completar y la información que se le pide se puede encontrar en las hojas de características del microcontrolador. Esta plantilla se puede encontrar en la misma ruta en la que se encuentran el resto de archivos que conforman la HAL. Ver Figura 32.

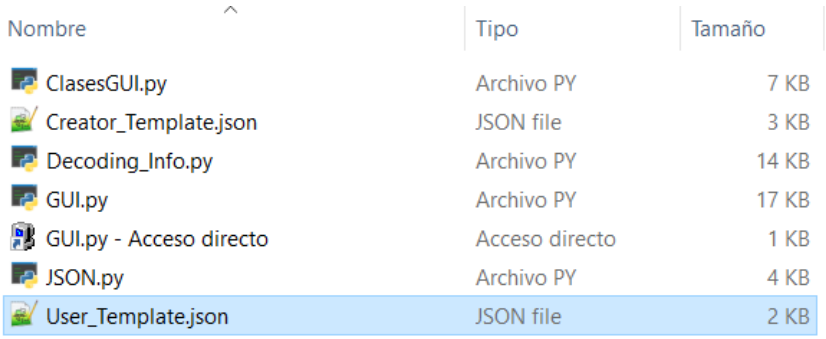


Figura 32. Plantilla User_Template.json a modificar por el programador. *[24]*

En esta plantilla se encontrará la información referente al nombre del microcontrolador con el que se está trabajando, al número de periféricos que incorpora cada microcontrolador y la información contenida en el archivo Vital_Information.h visto en capítulos anteriores. Ver Figura 33.

```
"Microcontroller_Name" : "RSL10",
"Microcontroller_Interfaces": [
                                {
                                    "Interface_name": "GPIO",
                                    "Ports_In_The_Microcontroller":8,
                                    "Pines_Per_Port": 8
                                },

                                {
                                    "Interface_name": "Timer",
                                    "Number_Of_Timer_Interfaces": 15
                                },
                                {
                                    "Interface_name": "SPI",
                                    "SPI_NUMBER_OF_INTERFACES": 4
                                },
                                {
                                    "Interface_name": "I2C",
                                    "I2C_NUMBER_OF_INTERFACES": 5
                                },
                                {
                                    "Interface_name": "UART",
                                    "NUMBER_OF_UART_INTERFACES": 2
                                },
                                {
                                    "Interface_name": "ADC",
                                    "NUMBER_OF_ADC_INTERFACES": 3,
                                    "TOP_REFERENCE_t" : ["REFERENCE_2V", "REFERENCE_3_3V"],
                                    "BOTTOM_REFERENCE_t" : ["REFERENCE_GND", "REFERENCE_GND_IN"]
                                },
```

Figura 33. Interior de la plantilla User_Template.json. *[24]*

Una vez completada esta plantilla, la Interfaz Gráfica se ejecutará adaptándose a características del microcontrolador generando una Interfaz personalizada.

Capítulo 4: Demostración

Se ha realizado un proyecto para que sirva de demostración del correcto funcionamiento de la HAL.
La demostración consiste en una comunicación entre una aplicación para dispositivos móviles y los dos microcontroladores vistos en el capítulo “2.3 Periféricos”, haciendo uso para ambos de las mismas funciones de la HAL, con lo que se podrá demostrar que estas funciones tienen el mismo nombre y se comportan de la misma forma de la misma forma independientemente del microcontrolador de trabajo.

La aplicación, de elaboración propia, realizará el envío de unos comandos, previamente establecidos, vía bluetooth a un módulo bluetooth que a su vez enviará su información a un microcontrolador para que este encienda y apague una lámpara o encienda un led rojo verde o azul según se indique desde la aplicación.

4.1 App Inventor:

App Inventor es un entorno de programación gráfico que permite crear aplicaciones para teléfonos móviles y tabletas Android. [25]

La programación está basada en bloques (ver Figura 35), que como si de un puzle se tratara, se pueden ir ensamblando para posteriormente generar un App que pueda ser ejecutado en un dispositivo Android. El entorno de desarrollo incluye un gran número de componentes gráficos (botones, etiquetas, deslizadores, selectores de lista), que pueden ser arrastrados a un visor, con forma de teléfono móvil, para visualizar la aplicación que se está desarrollando.

La aplicación desarrollada se puede ver en la Figura 34. Esta aplicación cuenta con tres botones, de colores rojo, verde y azul, un botón de encendido y apagado de una lámpara y un seleccionador de lista para elegir el módulo Bluetooth al que conectarse.
Cuando alguno de los botones rojo, verde o azul es pulsado se envía un comando vía bluetooth al microcontrolador, indicándole que encienda o apague uno de los leds a los que tiene acceso.
Cuando el botón de encendido y apagado de lámpara sea pulsado, se le enviará al microcontrolador otro comando distinto que le indique que debe encender o apagar según corresponda.
Finalmente, el seleccionador de lista es un botón que al ser pulsado muestra una lista de los dispositivos Bluetooth a los que el móvil puede conectarse.

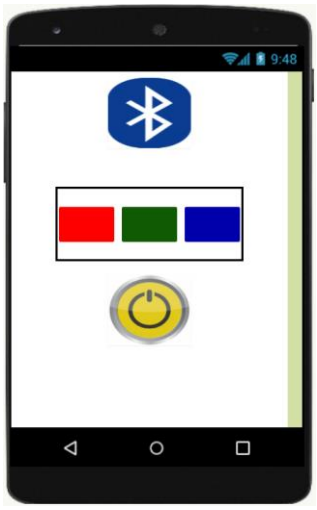


Figura 34. Vista de la aplicación móvil en el entorno de desarrollo MIT App Inventor.[24]

La programación de la aplicación se ha realizado mediante bloques como se puede ver en la Figura 35, en la que se muestran los bloques de código asociados a los botones rojo y azul.
Cuando se pulsa cualquiera de estos botones se evalúa su estado (si el led está encendido o no) y según sea este, se envía a través de bluetooth el comando de ejecución correspondiente (puede ser '3', el '6', el '1' o el '4') para que el microcontrolador actúe en consecuencial.

Figura 35. Bloques de código implementados en la pestaña de bloques de App Inventor.[24]

4.2 Componentes

Para realizar esta demostración se han utilizado los siguientes componentes: un módulo bluetooth HC-05 en cada microcontrolador para gestionar la comunicación bluetooth, un módulo de relé capaz de soportar corrientes de 10 Amperios (es igual que un magnetotérmicos de protección ante sobrecorrientes que se encuentran en los hogares de hoy en día) controlable con una señal digital para encender y apagar una lámpara y un módulo RGB (un módulo con tres leds rojo, verde y azul) para poder crear luces en base a esos tres colores. Ver Figura 36.

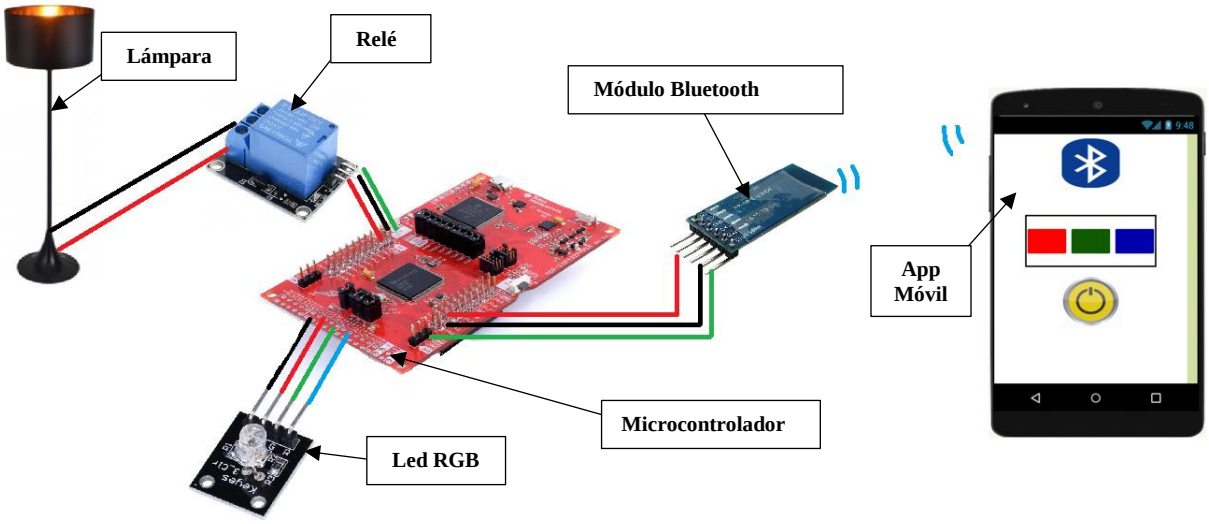


Figura 36. Elementos utilizados en la demostración.[24]

4.3 Código demostración

El código desarrollado para ambos microcontroladores se recoge en el Anexo I y Anexo II en el que se puede observar cómo todas las funciones necesarias para poder realizar esta demostración son las mismas para cada microcontrolador, verificando que la HAL funciona correctamente pues se ha conseguido independizar las funciones generales de control de cada periférico del microcontrolador que se esté utilizando en ese momento.

Puede apreciarse como las funciones y parámetros utilizados son iguales para ambos microcontroladores.
Por ejemplo, para la configuración de los pines en ambos casos se ha utilizado la función "*Dio_Configure(0, 2, DIO_NO_PULL_RESISTOR, DIO_OUTPUT, DIO_LOW, DIO_DRIVE_6X);*" y para leer información de la UART se ha utilizado *"Uart_Read_Interrupts(0,RxBufffer, sizeof(RxBufffer),BT_Reception);"* consiguiendo en ambos casos los mismos resultados en los dos microcontroladores.

En conclusión, la HAL realmente se comporta como debería, permitiendo a un programador hacer uso de las mismas funciones para configurar y controlar microcontroladores de distintos fabricantes.

Capítulo 5: Conclusiones

1. El proyecto ha concluido satisfactoriamente con el desarrollo en C de una capa de acceso al Hardware (HAL) portable para procesadores ARM Cortex-M, la cual permite controlar 6 periféricos como son DIO, SPI, ADC, UART, I2C, TIMERS.

2. Se ha logrado crear una Interfaz gráfica con Tkinter para facilitar el acceso a la capa HAL, la cual permite de forma configurar los periféricos de forma gráfica.
   Además, la interfaz permite crear un archivo con las funciones necesarias para la configuración de periféricos incluyendo las librerías necesarias para ello.

3. Se ha generado una documentación explicativa de los recursos de la HAL a través de Doxygen la cual se adjunta en el Anexo III.

4. Para la realización de este TFG se han utilizado protocolos estandarizados y consolidados en el mercado. Python como lenguaje de programación de la GUI, JSON como formato para organizar la información referente a un microcontrolador que posteriormente leerá la GUI, lenguaje C para la programación de microcontroladores y Doxygen como generador de documentación. Como se puede apreciar, se ha tenido en cuenta en todas las fases del proyecto que se estuviera utilizando tecnología estandarizada y común en la industria ofreciendo facilidades para iniciarse con la herramienta generada.

5. Como puede verse en el capítulo “2.6.7 Verificación Funcional” se ha seguido una metodología para la Verificación y corrección de errores de la HAL, evitando así posibles fallos que pudieran existir en la HAL.

6. Se han configurado las capas de Drivers para los dos microcontroladores con procesador Cortex ARM M (el MSP432P401R y el RSL10) verificando que la HAL cumple el propósito para el que ha sido diseñada.

7. Se ha justificado la viabilidad técnica de la HAL a través de una demostración que consta una aplicación móvil con App Inventor que se comunique a través de unos comandos previamente establecidos con un módulo bluetooth conectado a los dos microcontroladores de estudio. Estos microcontroladores han sido programados mediante la Interfaz Gráfica y la HAL desarrollada, verificando que las funciones que aparezcan en el código que será cargado en los microcontroladores es el mismo y generan el mismo comportamiento en ambos.

Capítulo 6: Líneas Futuras de desarrollo

Las líneas de investigación y desarrollo que podrían dar lugar a una mayor amplitud del proyecto y que debería tenerse en cuenta en posteriores actualizaciones del mismo son las siguientes:

1. **Optimización de código y memoria.** Este TFG se ha centrado principalmente en la funcionalidad sin preocuparse en cuantos recursos de memoria están siendo utilizados por lo que mejorar la eficiencia en el uso de memoria podría suponer una gran diferencia para poder utilizar aquellos microcontroladores con menos capacidades.

2. **Añadir nuevos periféricos.** Periféricos como el DMA, CAN, más funciones de los Timers, no han sido incorporados a la actual HAL y se encuentran en un gran número de microcontroladores por lo que incorporarlos a la actual tecnología añadirá una mayor funcionalidad a la actual HAL.

3. **Drivers para sensores.** Aunque una HAL no está obligada a dar soporte a los sensores actuales, incorporar el software necesario para los sensores más comunes haciendo uso de las funciones ya incorporadas puede ampliar las capacidades que ofrece esta tecnología.

4. **Mejorar la estética de la interfaz gráfica.** La interfaz gráfica ha sido creada con objeto de ser funcional sin preocuparse en exceso por la estética de la misma. Un mejor acabado podrá hacerla más visual para el usuario.

Capítulo 7: Bibliografía

[1] Autor: Jacob Beningo – Título: “Reusable Firmware Development: A Practical Approach to APIs, HALs and Drivers”

[2] Autor: Michael Barr – Título: “Zero Bugs...Embedded C Coding Standard”

[3] Autor: Joseph Yiu – Título: “The definitive guide to ARM Cortex M3 and Cortex M4 processors”

[4] Programación por capas. https://es.wikipedia.org/wiki/Programaci%C3%B3n_por_capas

[5] Definición de periférico. https://hetpro-store.com/TUTORIALES/microcontrolador/

[6] Microcontrolador MSP432P401R. https://www.ti.com/tool/MSP-EXP432P401R

[7] Autor: Texas Instruments - Título: “MSP432P401R, MSP432P401M SimpleLink™ Mixed-Signal Microcontrollers”

[8] SoC RSL10. https://www.onsemi.com/products/connectivity/wireless-rf-transceivers/rsl10

[9] Autor: ON Semiconductor – Título: “RSL10_hardware_reference”

[10] Interfaz Gráfica. https://es.wikipedia.org/wiki/Interfaz_gr%C3%A1fica_de_usuario

[11] Información básica de Python. https://es.wikipedia.org/wiki/Python

[12] Información básica Json. https://www.json.org/json-es.html

[13] Keil. http://www.keil.com/

[14] Introducción a CMSIS. http://www.keil.com/pack/doc/CMSIS/General/html/index.html

[15] Núcleo M CMSIS. http://www.keil.com/pack/doc/CMSIS/Core/html/index.html

[16] SVD CMSIS. http://www.keil.com/pack/doc/CMSIS/SVD/html/index.html

[17] Doxygen Introducción. https://en.wikipedia.org/wiki/Doxygen

[18] Lenguajes compatibles con Doxygen. https://www.doxygen.nl/index.html

[19] HAL. https://whatis.techtarget.com/definition/hardware-abstraction-layer-HAL

[20] ARM Cortex A. https://developer.arm.com/ip-products/processors/cortex-a

[21] ARM Cortex R. https://developer.arm.com/ip-products/processors/cortex-r

[22] ARM Cortex M. https://developer.arm.com/ip-products/processors/cortex-m

[23] Historia ARM. https://es.wikipedia.org/wiki/Arquitectura_ARM

[24] Fuente: Elaboración propia.

[25] App Inventor: https://appinventor.mit.edu/about-us

[26] Tkinter: https://wiki.python.org/moin/TkInter

Capítulo 8: Anexos

Anexo I: Código demostración RSL10

```
#include <stdint.h>
#include "GPIO.h"
#include "UART.h"
#include "Clock.h"

char RxBufffer[1];
void BT_Reception(void);
void Hardware_Configuration(void);

int main(void)
{
	//Configuring Hardware
	Hardware_Configuration();

	//Reading function:
	Uart_Read_Interrupts(0,RxBufffer, sizeof(RxBufffer),BT_Reception);

	while(1)
	{
		//Disabling WDT
		Sys_Watchdog_Refresh();  // Refresh watchdog timer
	}
}


void Hardware_Configuration(void)
{
	//Enable global Interrupt:
	Sys_NVIC_ClearAllPendingInt();
	Sys_NVIC_DisableAllInt();
	__set_PRIMASK(PRIMASK_ENABLE_INTERRUPTS);

		//Configurat Clocks:
	Configure_Oscillator(RC_3_TO_12_MHZ, 3000000);
	Configure_Clock_Feeded_By_Oscillators(RCCLK, RC_3_TO_12_MHZ, 3000000);
	Configure_Mainclock(RCCLK, 3000000);
	Configure_Clock_Derived_From_Mainclock(UARTCLK, 9600);

	/*
	*********************************************************************************
	********************* DIO  CONFIGURATION  ***************************************
	*********************************************************************************
	*/

	/*Configurando pin:	P0.4
	Resistencia interna:	 DIO_NO_PULL_RESISTOR
	Direccion del pin:	DIO_OUTPUT
	Estado inicial:		DIO_LOW
	Drive:			DIO_DRIVE_6X
	*/
	Dio_Configure(0, 4, DIO_NO_PULL_RESISTOR, DIO_OUTPUT, DIO_LOW, DIO_DRIVE_6X );
```

```
    /*Configurando pin:     P0.5
      Resistencia interna:  DIO_NO_PULL_RESISTOR
      Direccion del pin:    DIO_OUTPUT
      Estado inicial:       DIO_LOW
      Drive:                DIO_DRIVE_6X
    */
    Dio_Configure(0, 5, DIO_NO_PULL_RESISTOR, DIO_OUTPUT, DIO_LOW, DIO_DRIVE_6X );


    /*Configurando pin:     P0.6
      Resistencia interna:  DIO_NO_PULL_RESISTOR
      Direccion del pin:    DIO_OUTPUT
      Estado inicial:       DIO_LOW
      Drive:                DIO_DRIVE_6X
    */
    Dio_Configure(0, 6, DIO_NO_PULL_RESISTOR, DIO_OUTPUT, DIO_LOW, DIO_DRIVE_6X );

    /*Configurando pin:     P0.2
      Resistencia interna:  DIO_NO_PULL_RESISTOR
      Direccion del pin:    DIO_OUTPUT
      Estado inicial:       DIO_LOW
      Drive:                DIO_DRIVE_6X
    */
    Dio_Configure(0, 2, DIO_NO_PULL_RESISTOR, DIO_OUTPUT, DIO_LOW, DIO_DRIVE_6X );

    /*Configurando pin:     P0.1
      Resistencia interna:  DIO_NO_PULL_RESISTOR
      Direccion del pin:    DIO_OUTPUT
      Estado inicial:       DIO_HIGH
      Drive:                DIO_DRIVE_6X
    */
    Dio_Configure(0, 1, DIO_NO_PULL_RESISTOR, DIO_OUTPUT, DIO_HIGH, DIO_DRIVE_6X );

    /*****************************************************************************
    ************************** UART CONFIGURATION ********************************
    *****************************************************************************
    */

    /* Configurando la Interfaz UART: 0
       Pin RX:                              P0.0
       Pin TX:                              P0.1
       Baudios:                             UART_BAUD_RATE_9600
       Formato de datos:                    UART_DATA_FORMAT_8
       MSB First:                           UART_MSB_FIRST
       Paridad:                             UART_NO_PARITY
    */
    Uart_Configure(0, 0, 0 ,1, 0, UART_BAUD_RATE_9600, UART_NO_STOP_BIT,
                   UART_DATA_FORMAT_8, UART_MSB_FIRST, UART_NO_PARITY);

}
```

```
void BT_Reception(void)
{
	if(RxBufffer[0] == '1')
	{
		//Encender led rojo
		Dio_Set_High(0,6);

	}
	else if(RxBufffer[0] == '4')
	{
		//Apagar led rojo
		Dio_Set_Low(0,6);

	}
	else if(RxBufffer[0] == '2')
	{
		//Encender led verde
		Dio_Set_High(0,5);

	}
	else if(RxBufffer[0] == '5')
	{
		//Apagar led rojo
		Dio_Set_Low(0,5);

	}
	else if(RxBufffer[0] == '3')
	{
		//Encender led azul
		Dio_Set_High(0,4);

	}
	else if(RxBufffer[0] == '6')
	{
		//Apagar led azul
		Dio_Set_Low(0,4);

	}
	else if(RxBufffer[0] == '7')
	{
		//Encender Lampara
		Dio_Set_High(0,2);

	}
	else if(RxBufffer[0] == '8')
	{
		//Apagar Lampara
		Dio_Set_Low(0,2);

	}
	Uart_Read_Interrupts(0,RxBufffer, sizeof(RxBufffer),BT_Reception);
}
```

Anexo II Código Demostración MSP432

```
#include <stdint.h>
#include "msp432p401r.h"
#include "GPIO.h"
#include "UART.h"
#include "Clock.h"

char RxBufffer[1];
void BT_Reception(void);
void Hardware_Configuration(void);

int main(void)
{
      Hardware_Configuration();
      //Reading function:
      Uart_Read_Interrupts(0,RxBufffer, sizeof(RxBufffer),BT_Reception);
      while(1);

}

void Hardware_Configuration(void)
{
      /*ENABLE GLOBAL INTERRUPT*/
     __enable_irq();

      //Disabling WDT
      WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;  // Stopping watchdog timer

      /*************************************************************************
      ********************* CLOCK CONFIGURATION  ***************************
      *************************************************************************/
      //Configuring oscillator:
      Configure_Oscillator(DCO_OSCILLATOR,12000000);

      //Configuring MCLK:
      Configure_Clock_Feeded_By_Oscillators(MCLK,DCO_OSCILLATOR , 12000000 );

      //Configuring SMCLK:
      Configure_Clock_Feeded_By_Oscillators(SMCLK, DCO_OSCILLATOR ,12000000 );

/*****************************************************************************
***************************** UART CONFIGURATION ******************************
*****************************************************************************/

      /*Configurando la Interfaz UART: 0
        Pin RX:                                 P2.2
        Pin TX:                                 P2.3
        Baudios:                                UART_BAUD_RATE_9600
        Formato de datos:                 UART_DATA_FORMAT_8
        MSB First:                              UART_LSB_FIRST
        Paridad:                                UART_NO_PARITY
      */
      Uart_Configure(0, 2, 2 ,3, 2, UART_BAUD_RATE_9600, UART_ONE_STOP_BIT,
                     UART_DATA_FORMAT_8, UART_LSB_FIRST, UART_NO_PARITY);
```

```
/*****************************************************************************************
********************************** DIO  CONFIGURATION ************************************
*****************************************************************************************/

      /*Configurando pin:     P6.4
        Resistencia interna:  DIO_STRONG_PULL_UP_RESISTOR
        Direccion del pin:    DIO_INPUT
        Estado inicial:       DIO_STATE_NOT_CONFIGURED
        Drive:                DIO_DRIVE_NOT_CONFIGURED
      */
      Dio_Configure(5, 4, DIO_RESISTOR_NOT_CONFIGURED, DIO_OUTPUT, DIO_HIGH, DIO_DRIVE_6X );

      /*Configurando pin:     P2.0
        Resistencia interna:  DIO_RESISTOR_NOT_CONFIGURED
        Direccion del pin:    DIO_OUTPUT
        Estado inicial:       DIO_HIGH
        Drive:                DIO_DRIVE_6X
      */
      Dio_Configure(1, 0, DIO_RESISTOR_NOT_CONFIGURED, DIO_OUTPUT, DIO_LOW, DIO_DRIVE_6X );


      /*Configurando pin:     P2.1
        Resistencia interna:  DIO_RESISTOR_NOT_CONFIGURED
        Direccion del pin:    DIO_OUTPUT
        Estado inicial:       DIO_LOW
        Drive:                DIO_DRIVE_6X
      */
      Dio_Configure(1, 1, DIO_RESISTOR_NOT_CONFIGURED, DIO_OUTPUT, DIO_LOW, DIO_DRIVE_6X );


      /*Configurando pin:     P2.2
        Resistencia interna:  DIO_RESISTOR_NOT_CONFIGURED
        Direccion del pin:    DIO_OUTPUT
        Estado inicial:       DIO_LOW
        Drive:                DIO_DRIVE_6X
      */
      Dio_Configure(1, 2, DIO_RESISTOR_NOT_CONFIGURED, DIO_OUTPUT, DIO_LOW, DIO_DRIVE_6X );

}
```

```
void BT_Reception(void)
{
	if(RxBufffer[0] == '1')
	{
		//Encender led rojo
		Dio_Set_High(1,0);
	}
	else if(RxBufffer[0] == '2')
	{
		//Encender led verde
		Dio_Set_High(1,1);
	}
	else if(RxBufffer[0] == '3')
	{
		//Encender led azul
		Dio_Set_High(1,2);
	}
	else if(RxBufffer[0] == '4')
	{
		//Apagar led rojo
		Dio_Set_Low(1,0);
	}
	else if(RxBufffer[0] == '5')
	{
		//Apagar led verde
		Dio_Set_Low(1,1);
	}
	else if(RxBufffer[0] == '6')
	{
		//Apagar led azul
		Dio_Set_Low(1,2);
	}
	else if(RxBufffer[0] == '7')
	{
		//Encender Lampara
		Dio_Set_High(5,4);
	}
	else if(RxBufffer[0] == '8')
	{
		//Apagar Lampara
		Dio_Set_Low(5,4);
	}
	Uart_Read_Interrupts(0,RxBufffer, sizeof(RxBufffer),BT_Reception);
}
```

Anexo III Documentación Doxygen

1. Interfaz DIO

◆ Dio_Configure()

```
DRIVER_RETURN_CODES_t Dio_Configure ( const uint8_t          _port,
                                      const uint8_t          _channel,
                                      const DIO_RESISTOR_t   _resistor,
                                      const DIO_DIRECTION_t  _direction,
                                      const DIO_PIN_STATE_t  _state,
                                      const DIO_DRIVE_t      _drive
                                    )
```

Description: This function is used to configure a channel as a Digital Input/Output pin.

Parameters

_port	Port of the pin to be set as High
_channel	Channel to be set as High
_resistor	Channel to be set as High
_direction	Channel to be set as High
_state	Channel to be set as High
_drive	Channel to be set as High

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
//Configuring pin: P2.1
//Internal resistor: DIO_NO_PULL_RESISTOR
//Direccion del pin: DIO_OUTPUT
//Estado inicial: DIO_HIGH
//Drive: DIO_DRIVE_5X

Dio_Configuration(2,1,DIO_NO_PULL_RESISTOR , DIO_OUTPUT, DIO_HIGH, DIO_DRIVE_5X );
```

◆ Dio_Read_Channel()

```
uint32_t Dio_Read_Channel ( const uint8_t  _port,
                            const uint8_t  _channel
                          )
```

Description: This function is used to read whether the sate of the selected channel is High or Low.

Parameters

_port Port of the pin we want to read

_channel Channel that is going to be read

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
int main(void)
{
    //Reading pin: P2.1
    uint8_t portRead;
    pinRead = Dio_Read_Channel(2,1);
    if(pinRead == 1)
    {
        // Do something when portRead == 1
    }
    else
    {
        // Do something when portRead == 0
    }
}
```

◆ Dio_Read_Port()

uint32_t Dio_Read_Port (const uint8_t _port)

Description: This function is used to read whether the sate of all the pins from the selected port.

Parameters

_port Port we want to read

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
int main(void)
{
  //Reading pin: P2.1
  uint8_t portRead;
  portRead = Dio_Read_Port(2);
  if(portRead == 0x4)
  {
      // Do something when portRead == 1
  }
  else
  {
      // Do something when portRead == 0
  }
}
```

◆ Dio_Set_High()

DRIVER_RETURN_CODES_t Dio_Set_High (const uint8_t _port,
const uint8_t _channel
)

Description: This function is used to set the state of the pin as High.

Parameters

_port Port of the pin we want to set as High

_channel channel number of the pin we want to set as High.

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
int main(void)
{
  //Pin to be set as High: P2.1
  Dio_Set_High(2,1);
}
```

◆ Dio_Set_Low()

DRIVER_RETURN_CODES_t Dio_Set_Low (const uint8_t _port,
const uint8_t _channel
)

Description: This function is used to set the state of the pin as Low.

Parameters

_port Port of the pin we want to set as Low

_channel channel number of the pin we want to set as Low.

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
int main(void)
{
  //Pin to be set as Low: P2.1
  Dio_Set_Low(2,1);
}
```

◆ Dio_Toggle()

DRIVER_RETURN_CODES_t Dio_Toggle (const uint8_t _port,
const uint8_t _channel
)

Description: This function is used to toggle the sate of the pin selected.

Parameters

_port Port we want to read

_channel Channel we want to read

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
int main(void)
{
  //Toggling the state pin: P2.1
  Dio_Toggle(2,1);
}
```

2. Interfaz Temporizadores

◆ Get_Timer_Current_Value()

uint32_t Get_Timer_Current_Value (const **INTERFACE_NUMBER_t** Interface_Number)

Description: This function gets the current countdown value.

Parameters

_interface_number number of the Timer interface

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
//Getting the current value of the timer countdown
Timer_Restart(0);
```

◆ Timer_Configure()

```
DRIVER_RETURN_CODES_t Timer_Configure ( const INTERFACE_NUMBER_t        _interface_number,
                                        const TIME_MS_t                 _time,
                                        const uint32_t                  _freq_of_clock_that_feeds_Timer,
                                        const TIMER_INTERRUPT_ENABLE_t  _interrupt_enable
                                      )
```

Description: This function configures one timer interface.

Parameters

_interface_number	number of the Timer interface
_time	time in miliseconds before triggering the timer
_freq_of_clock_that_feeds_Timer	Clock of the frequency that feeds the timer
_interrupt_enable	flag that indicates whether the Timer is expected to trigger an interrupt when time passes or not

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
//Configurando Timer:                                       0
//Tiempo a contar [ms]:                                     1000
//Frecuencia de trabajo del reloj que alimenta al Timer [kHz]:  3000000
//Interrupcion Estado:                                      TIMER_INTERRUPT_ENABLE
Timer_Configure( 0, 1000, 3000000 ,TIMER_INTERRUPT_ENABLE);
```

◆ Timer_Restart()

DRIVER_RETURN_CODES_t Timer_Restart (const **INTERFACE_NUMBER_t** Interface_Number)

Description: This function restarts the countdown.

Parameters

_interface_number number of the Timer interface

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
//Restarting the Timer countdown from the interface number 0
Timer_Restart(0);
```

◆ Timer_Start()

DRIVER_RETURN_CODES_t Timer_Start (const **INTERFACE_NUMBER_t** Interface_Number)

Description: This function starts the countdown.

Parameters

_interface_number number of the Timer interface

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
//Starting the Timer countdown from the interface number 0
Timer_Start(0);
```

◆ Timer_Stop()

DRIVER_RETURN_CODES_t Timer_Stop (const **INTERFACE_NUMBER_t** Interface_Number)

Description: This function stops the countdown.

Parameters

_interface_number number of the Timer interface

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
//Stoping the Timer countdown from the interface number 0
Timer_Stop(0);
```

3. Interfaz UART

◆ Uart_Write()

```
DRIVER_RETURN_CODES_t Uart_Write ( const INTERFACE_NUMBER_t  _interface_number,
                                   void *                    pTxBuffer,
                                   uint32_t                  Len
                                 )
```

Description: This function writes over Uart Interface but this interface do not use the interrupts, so this is a blocking function.

Parameters

_interface_number	number of the Uart interface
pTxBuffer	Pointer to the bufferof bytes to be sent
Len	size of bytes to be sent

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
//Definition of one array that has the chararcters to be sent
char array = "Menuda HAL";
//This function will Write over Uart
// Parameters:   Uart Interface number
//               Pointer to buffer
//               Number of bytes to be sent to be sent
Uart_Write(0, array, sizeof(array));
```

◆ Uart_Write_Interrupts()

DRIVER_RETURN_CODES_t Uart_Write_Interrupts (const **INTERFACE_NUMBER_t** _interface_number,
void * pTxBuffer,
uint32_t Len,
tx_has_finished_callback_function_t _cb
)

Description: This function writes over Uart Interface and this interface uses interrupts.

Parameters

_interface_number	number of the Uart interface
pTxBuffer	Pointer to the bufferof bytes to be sent
Len	size of bytes to be sent
_cb	transmission callback function to be called when all bytes are sent.

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
//Definition of one array that has the chararcters to be sent
char array = "Menuda HAL";
//This function will Write over Uart
// Parameters:   Uart Interface number
//               Pointer to buffer
//               Number of bytes to be sent to be sent
Uart_Write_Interrupts(0, array, sizeof(array));
```

◆ Uart_Tx_Interrupt_Handler()

DRIVER_RETURN_CODES_t Uart_Tx_Interrupt_Handler (const **INTERFACE_NUMBER_t** _interface_number)

Description: This funtion is the Handler that when a Uart transmission of one byte is completed and the interrupt is triggered, the User has to call it passing to it the interface number as parameter. This function will continue the transmission until it reaches the number of bytes specified in Uart_Write_Interrupts.

Parameters

_interface_number number of the Uart interface

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
//In the real Handler for the Uart transmision interrupt; for example, Uart_Handler_Interface1() the user has to call this function.
Uart_Handler_Interface1()
{
    Uart_Uart_Read_Interrupts(0);
}
```

◆ Uart_Rx_Interrupt_Handler()

DRIVER_RETURN_CODES_t Uart_Rx_Interrupt_Handler (const **INTERFACE_NUMBER_t** _interface_number)

Description: This funtion is the Handler that when a Uart reception of one byte is completed and the interrupt is triggered, the User has to call it passing to it the interface number as parameter. This function will continue the reception until it reaches the number of bytes specified in Uart_Read_Interrupts.

Parameters

_interface_number number of the Uart interface

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
//In the real Handler for the Uart transmision interrupt; for example, Uart_Handler_Interface1() the user has to call this function.
Uart_Handler_Interface1()
{
    Uart_Uart_Read_Interrupts(0);
}
```

◆ Uart_Read_Interrupts()

DRIVER_RETURN_CODES_t Uart_Read_Interrupts (const **INTERFACE_NUMBER_t** _interface_number,
void * pRxBuffer,
uint32_t Len,
rx_has_finished_callback_function_t _cb
)

Description: This funtion will read from the UART. This function do use Interrupts. This is a non blocking function.

Parameters

_interface_number	number of the Uart interface
pRxBuffer	Pointer to the bufferof bytes to be sent
Len	size of bytes to be sent
rx_has_finished_callback_function_t	callback function to be called when all bytes expected to be received arrive.

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
//Definition of one buffer to store the data coming
char array[10];
//This function will Read from Uart Interface
// Parameters:   Uart Interface number
//               Pointer to buffer to store the data
//               Number of bytes expected to be received
Uart_Read(0, array, sizeof(array));
```

◆ Uart_Read()

DRIVER_RETURN_CODES_t Uart_Read (const **INTERFACE_NUMBER_t** _interface_number,
void * pRxBuffer,
uint32_t Len,
uint32_t _delay_cycles_waiting
)

Description: This funtion will read from the UART. This function do not use Interrupts. This is a blocking functions.

Parameters

_interface_number	number of the Uart interface
pRxBuffer	Pointer to the bufferof bytes to be sent
Len	size of bytes to be sent
_delay_cycles_waiting	number of cycles to be waiting during the reception without receiving any data. If no data is

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
//Definition of one buffer to store the data coming
char array[10];
//This function will Read from Uart Interface
// Parameters:   Uart Interface number
//               Pointer to buffer to store the data
//               Number of bytes expected to be received
Uart_Read(0, array, sizeof(array));
```

◆ Uart_Configure()

```
DRIVER_RETURN_CODES_t Uart_Configure ( const INTERFACE_NUMBER_t _interface_number,
                                       const RX_PIN_t          _rx_pin,
                                       const uint8_t           _rx_port,
                                       const TX_PIN_t          _tx_pin,
                                       const uint8_t           _tx_port,
                                       const uint32_t          _baudios,
                                       const UART_STOP_BITS_t  _stop_bits,
                                       const uint8_t           _data_bits_number,
                                       const uint8_t           _msb_first,
                                       const UART_PARITY_t     _parity
                                     )
```

Description: This function is used to configure the Uart Interface.

Parameters

_interface_number	Port we want to read
_rx_pin	Channel of the RX pin
_rx_port	Port of the RX pin
_tx_pin	Channel of the TX pin
_tx_port	Port of the TX pin
_baudios	Baud Rate for Uart Interface
_stop_bits	Number of stop bits
_data_bits_number	Number of data bits
_msb_first	MSB first
_parity	Parity of the Uart Interface

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
// Configurando la Interfaz UART: 0
// Pin RX: P1.0
// Pin TX: P3.2
// Baudios: UART_BAUD_RATE_9600
// Formato de datos: UART_DATA_FORMAT_8
// MSB First: UART_MSB_FIRST
// Paridad: UART_ODD_PARITY

Uart_Configure( 0, 0, 1 ,2, 3, UART_BAUD_RATE_9600, UART_TWO_STOP_BIT, UART_DATA_FORMAT_8, UART_MSB_FIRST, UART_ODD_PARITY);
```

4. Interfaz SPI

◆ Spi_Configure()

```
DRIVER_RETURN_CODES_t Spi_Configure ( const SPI_INTERFACE_NUMBER_t      _interface_number,
                                      const SPI_ROLE_t                  _role,
                                      const SPI_BUS_CONFIGURATION_t     _bus_configuration,
                                      const SPI_BUS_DATA_FORMAT_t       _data_format,
                                      const SPI_CPOL_t                  _cpol,
                                      const SPI_CPHA_t                  _cpha,
                                      const SPI_CLOCK_FRECUENCY_HZ_t    _spi_clock_frecuency,
                                      const MOSI_PIN_NUMBER_t           _mosi,
                                      const MOSI_PIN_NUMBER_t           _mosi_port,
                                      const MISO_PIN_NUMBER_t           _miso,
                                      const MOSI_PIN_NUMBER_t           _miso_port,
                                      const NSS_PIN_NUMBER_t            _nss,
                                      const MOSI_PIN_NUMBER_t           _nss_port,
                                      const SCK_PIN_NUMBER_t            _sck,
                                      const MOSI_PIN_NUMBER_t           _sck_port
                                    )
```

Description: This function configures one timer interface.

Parameters

_interface_number	
_role	Defines de role of the SPI interface
_bus_configuration	Defines the bus configuration
_data_format	Defines the data format of the bus
_cpol	Defines the CPOL parameter of the interface
_cpha	Defines the CPHA parameter of the interface
_spi_clock_frecuency	Defines the Clock frequency of the interface
_mosi	Defines MOSI pin
_mosi_port	Defines MOSI port
_miso	Defines MISO pin
_miso_port	Defines MISO port
_nss	Defines NSS pin
_nss_port	Defines NSS port
_sck	Defines SCK pin
_sck_port	Defines SCK port

Note

Example:

```
//Configurando la Interfaz SPI:   0
//Role: SPI_MASTER_MODE
//Bus Configuracion: SPI_FULL_DUPLEX
//Formato de los datos: SPI_DATA_FOTMAT_8_BITS
//CPOL: SPI_CPOL_0
//CPHA: SPI_CPHA_0
//Frecuencia del SPI clock[Hz]: 3000000
//Pin MOSI: P3.2
//Pin MISO: P1.0
//Pin NSS:    P5.4
//Pin SCK:    P7.6

Spi_Configured( 0, SPI_MASTER_MODE, SPI_FULL_DUPLEX ,SPI_DATA_FOTMAT_8_BITS, SPI_CPOL_0, SPI_CPHA_0, 3000000, 0, 1, 2, 3, 4 ,5 ,6, 7 );
```

◆ Spi_Read_Actual_Value_From_Buffer()

```
DRIVER_RETURN_CODES_t Spi_Read_Actual_Value_From_Buffer ( const SPI_INTERFACE_NUMBER_t _interface_number,
                                                          void *                       pRxBuffer
                                                        )
```

Description: This function configures one timer interface.

Parameters

_interface_number	number of the Timer interface
pRxBuffer	time in miliseconds before triggering the timer

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
//Definition of the transmission buffer to be sent
uint8_t rx_buffer;
//Configurando SPI interface: 0
//Buffer: tx_buffer
//Number of Bytes to be sent: sizeof(tx_buffer)
rx_buffer = Spi_Read_Actual_Value_From_Buffer(0 ,tx_buffer, sizeof(tx_buffer));
if(rx_buffer == 'a')
{
    //Do something here
}
```

◆ Spi_Read_Sending_Dummy_Bytes_Before_The_Read()

```
DRIVER_RETURN_CODES_t Spi_Read_Sending_Dummy_Bytes_Before_The_Read ( const SPI_INTERFACE_NUMBER_t _interface_number,
                                                                    void *                       pRxBuffer,
                                                                    uint32_t                     Len
                                                                  )
```

Description: This function reads over the SPI interface by sending the necessary bytes to receive the data.

Parameters

_interface_number	number of the Timer interface
pRxBuffer	time in miliseconds before triggering the timer
Len	Number of bytes to be set

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
//Definition of the transmission buffer to be sent
char rx_buffer[] = "Hola HAL";
//Configurando SPI interface: 0
//Buffer: rx_buffer
//Number of Bytes to be sent: sizeof(tx_buffer)
Spi_Read_Sending_Dummy_Bytes_Before_The_Read(0 ,rx_buffer, sizeof(rx_buffer));
```

◆ Spi_Write()

```
DRIVER_RETURN_CODES_t Spi_Write ( const SPI_INTERFACE_NUMBER_t _interface_number,
                                  void *                      pTxBuffer,
                                  uint32_t                    Len
                                )
```

Description: This function writes over the SPI interface.

Parameters

_interface_number	number of the Timer interface
pRxBuffer	time in miliseconds before triggering the timer
Len	Number of bytes to be set

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
//Definition of the transmission buffer to be sent
char tx_buffer[] = "Hola HAL";
//Configurando SPI interface: 0
//Buffer: tx_buffer
//Number of Bytes to be sent: sizeof(tx_buffer)
Spi_Write(0 ,tx_buffer, sizeof(tx_buffer));
```

5. Interfaz ADC

◆ Adc_Configure()

```
DRIVER_RETURN_CODES_t Adc_Configure ( const ADC_INTERFACE_NUMBER_t _adc_interface_number,
                                      const TOP_REFERENCE_t        _top_reference,
                                      const BOTTOM_REFERENCE_t     _bottom_reference,
                                      const SAMPLES_PER_SECOND_t   _samples_per_seconds,
                                      const uint8_t                _port,
                                      const uint8_t                _pin,
                                      const uint32_t               _freq_clock_that_feed_adc,
                                      const ADC_INTERRUPT_t        _interrupt_status
                                    )
```

Description: This function configures one ADC Interface.

Parameters

_adc_interface_number	Number of the ADC interface
_top_reference	This parameter sets the top reference of the ADC.
_bottom_reference	This parameter sets the bottom reference of the ADC.
_samples_per_seconds	This parameter sets the number of samples per second of the ADC
_port	This parameter indicates the port from which the ADC has to read.
_pin	This parameter indicates the pin from which the ADC has to read.
_freq_clock_that_feed_adc	This parameter indicates the frequency of the clock from which the ADC is feed.
_interrupt_status	This parameter indicates wheter the Interrupts of ADC should beenabled or not.

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
int main(void)
{

    //  ADC  CONFIGURATION
    //  Configurando ADC:                       0
    //  Top Reference Voltage:                  REFERENCE_3_3V
    //  Bottom Reference Voltage:               REFERENCE_GND
    //  Muestras por segundo:                   1000
    //  Pin Analogico:                          P1.2
    //  Frecuencia Reloj que alimenta al ADC:   1000000
    //  Interrupciones Estado:                  ADC_INTERRUPT_DISABLE

    Adc_Configure( 0, REFERENCE_3_3V, REFERENCE_GND, 1000, 1, 2 ,1000000, ADC_INTERRUPT_DISABLE);
}
```

◆ Adc_Conversion_Ended_Flag()

uint8_t Adc_Conversion_Ended_Flag (const **ADC_INTERFACE_NUMBER_t** _adc_interface_number)

Description: This function notifies the current state of one ADC Interface.

Parameters

_adc_interface_number This is the number of the interface that is going to be started

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
int main(void)
{
    //Waiting for the ADC to be free:
    while(Adc_Conversion_Ended_Flag(0) == ADC_IS_BUSY)
    {
        //TODO: Do things here.
    }
}
```

◆ Adc_Read()

uint32_t Adc_Read (const **ADC_INTERFACE_NUMBER_t** _adc_interface_number)

Description: This function reads one ADC Interface.

Parameters

_adc_interface_number This is the number of the interface that is going to be stopped

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
int main(void)
{
    //Reading from ADC interface number 2
    if(Adc_Read(2) > 500)
    {
        //TODO: Do things here
    }

}
```

◆ Adc_Start()

DRIVER_RETURN_CODES_t Adc_Start (const ADC_INTERFACE_NUMBER_t _adc_interface_number)

Description: This function Starts one ADC Interface.

Parameters

_adc_interface_number This is the number of the interface that is going to be started

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
int main(void)
{
    //Starting ADC Interface:
    Adc_Start(0);
}
```

◆ Adc_Stop()

DRIVER_RETURN_CODES_t Adc_Stop (const ADC_INTERFACE_NUMBER_t _adc_interface_number)

Description: This function stops one ADC Interface.

Parameters

_adc_interface_number This is the number of the interface that is going to be stopped

Returns

DRIVER_OK: If everything went well

DRIVER_ERROR_FROM_USER_LAYER: If something went wrong in

DRIVER_ERROR_PARAMETER: If any parameter is not correct

Note

Example:

```
int main(void)
{
    //Stopping ADC interface number 2
    Adc_Stop(2);
}
```